Python.industria(4.0)

PY006
Programación orientada a objetos
– Parte I
Comenzamos en 5’

Python.A_programar!

Ejercicios:

Fácil – Escalado de analógicas:

Escribe una función llamada "escalar_valores" que reciba un valor analógico como argumento y lo escale linealmente según un rango de entrada y un rango de salida específicos. El objetivo es transformar el valor analógico desde el rango de entrada al rango de salida.

valor_analogico: El valor analógico a escalar.
minimo_ing: El valor mínimo del rango de entrada. Default = 0
maximo_ing: El valor máximo del rango de entrada. Default = 32767
minimo: El valor mínimo del rango de salida. Default = 0.0
maximo: El valor máximo del rango de salida. Default = 100.0

Python.A_programar!

Ejercicios:

Intermedio – Verificador de temperaturas:

Hacer una función que tome, de una lista de valores, las temperaturas registadas en un día. De esa lista, debe retornar:

Promedio de temperaturas
La temperatura más baja
La temperatura más alta
Cantidad de temperaturas menores a 22°c

Hacer otra función que imprima los valores en un texto legible para el operador

```
Temperaturas1=[27.1, 22.3, 26.8, 23.5, 22.7, 15.3, 26.6, 16.9, 18.1, 24.7, 23.8, 18.4, 26.1, 27.5, 27.3, 21.9, 25.4, 25.1, 20.4, 16.2, 27.5, 22.7, 25.9, 21.2]
Temperaturas2=[25.4, 21.5, 27.3, 25.5, 20.2, 26.6, 16.1, 27.7, 26.4, 24.0, 22.6, 19.4, 27.0, 18.3, 25.0, 24.3, 25.6, 27.1, 15.6, 27.1, 26.6, 22.7, 20.4, 23.3]
Temperaturas3=[16.4, 20.5, 23.5, 17.3, 26.2, 26.2, 22.9, 21.2, 24.2, 26.0, 18.7, 27.5, 25.0, 22.7, 21.7, 22.7, 23.3, 25.0, 26.7, 18.7, 19.6, 23.9, 20.0, 17.2]
```

Python.industria(4.0)

PY006

Programación orientada a objetos – Parte I

Indice:

01(

Introducción

- Conceptos básicos
  - Objetos
  - Clases
  - Atributos
  - Métodos
- Ventajas

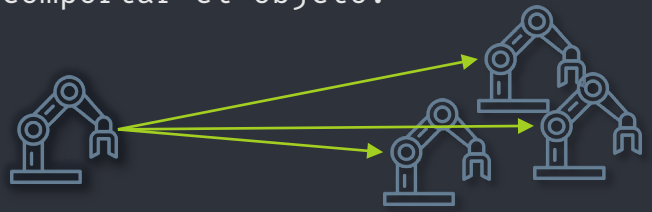
)

Class OOP:

La programación orientada a objetos es una metodología de programación, en la cual se organiza el código en objetos que interactúan entre sí.

En Python, los objetos son entidades que encapsulan datos, y comportamientos relacionados. Éstos objetos se crean a partir de plantillas denominadas clases, las cuales definen cómo es y cómo se debe comportar el objeto.

Clase: Brazo



Objeto: Brazo_1

Objeto: Brazo_2

Objeto: Brazo_3

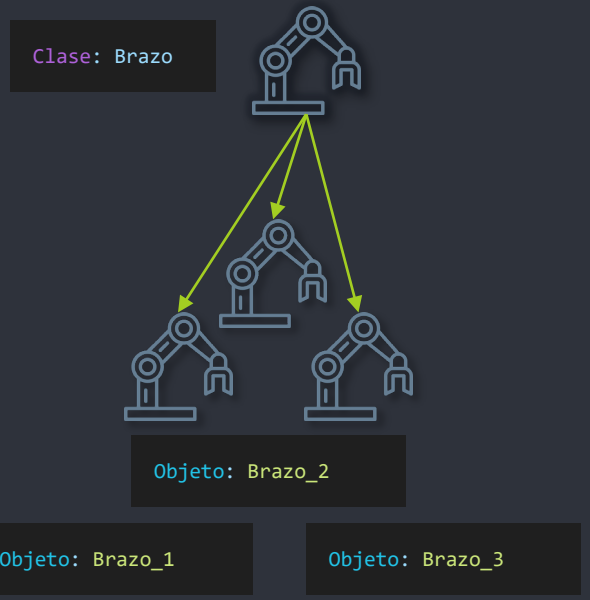
Es, ciertamente, un paradigma muy distinto a aquellos que están acostumbrados a la programación IEC, entonces, ¿cuáles son las ventajas que nos puede proveer ésta metodología de programación?

Class OOP:

Para responder ésta pregunta, veamos primero cuáles son las características y elementos clave de ésta metodología:

Elementos clave:
- Clases y atributos
- Objetos
- Métodos

Características:
- Encapsulación
- Herencia
- Polimorfismo



class Clase:

Las clases son estructuras que definen las propiedades y el comportamiento que tendrán los objetos.

Vienen a ser algo así como una plantilla, la cual utilizaremos después para “copiar” o “instanciar” cada objeto

Por supuesto, al definir una clase también definimos los atributos característicos de esa clase, aquellas propiedades únicas que pertenecen a ese objeto.

Las características pueden ser de entrada (las definimos al crear el objeto) o de salida (pueden modificarse con el correr del software)

Clase - Motor:

Entradas
- Potencia
- Voltaje
- Fabricante

Salidas
- Velocidad (RPM)

class Objeto:

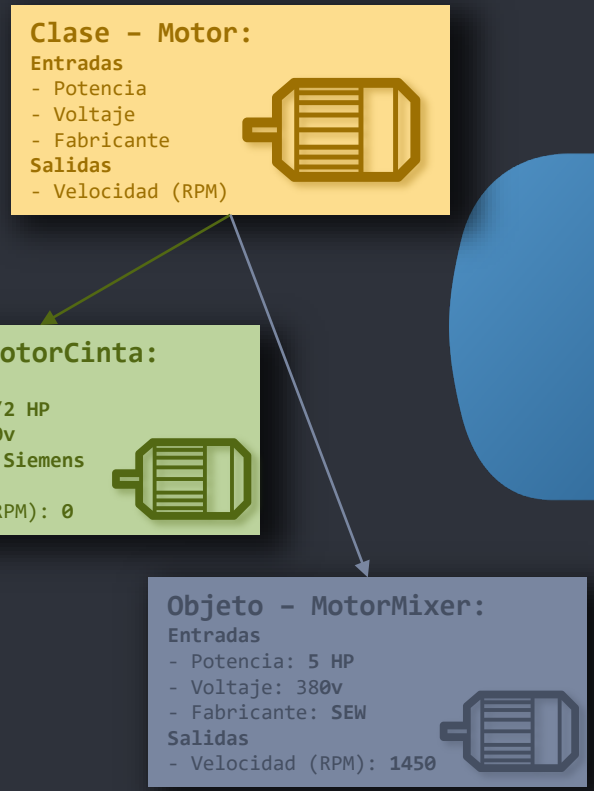
Una vez creada la clase principal, podremos instanciar cada uno de los objetos.

De esta manera, crearemos un elemento que posee todas las características que definimos en la clase principal.

Ese objeto tiene sus propias magnitudes, independientes de cualquier otro objeto que creemos.

Podemos acceder a cada uno de los elementos o atributos de un objeto con la notación de punto, como por ejemplo

```
MotorCinta.Fabricante
```



class método:

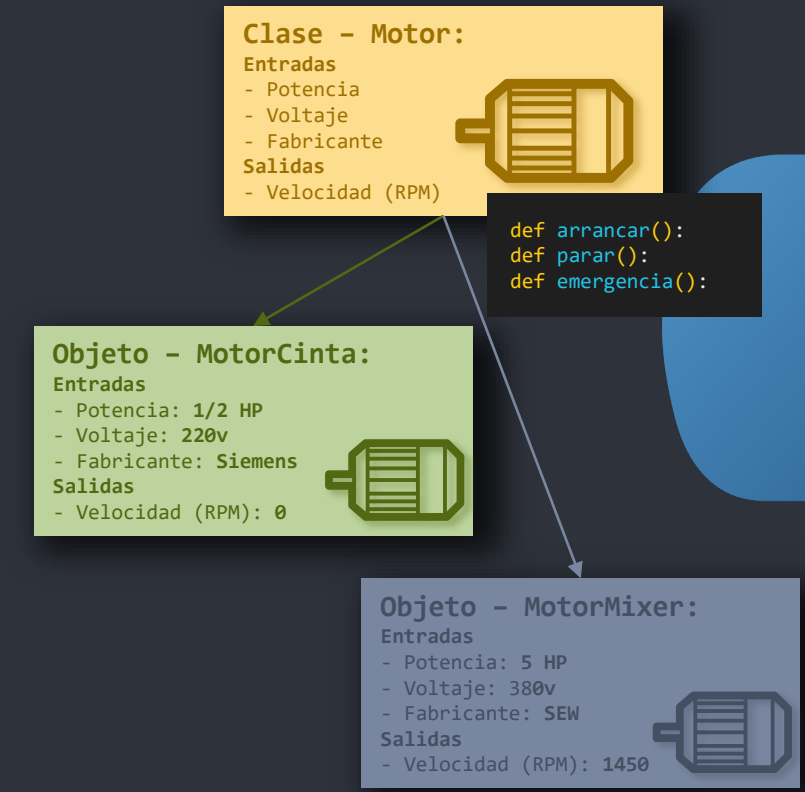
Los métodos en Python son una poderosa manera de organizar mi programa.

Un método es una función asociada a un objeto, que define su comportamiento.

Los métodos pueden acceder y modificar los atributos de un objeto, y pueden interactuar con otros objetos.

Los métodos se definen al momento de definir la clase, y se “llaman” con notación de punto. Por ejemplo:

```
MotorMixer.arrancar()
```



class encapsulación:

La encapsulación en la programación orientada a objetos, se refiere a la posibilidad de ocultar los detalles internos de un objeto.

De ésta manera, solo exponemos aquellos elementos esenciales a través de variables o métodos que llamamos “interfaz”

Por ejemplo, un sensor de temperatura puede encapsular la lógica de lectura y validación de datos internamente, exponiendo solo un método público para obtener la temperatura actual.

Clase – Temperatura:

Entradas
- Minimo
- Maximo
- Minimo_ing
- Maximo_ing
- Lectura sensor

Salidas
- Lectura escalada (°C)

```
def __leerPLC():
def __escalarTemp():
def __validarTemp():
def leerTemperatura():
```

```
SensorLaboratorio.leerTemperatura()
```

class herencia:

La herencia es una herramienta que permite crear clases nuevas en función a clases existentes.

De ésta manera, podemos crear distintas jerarquías y reutilizar métodos y atributos a lo largo de nuestro programa.

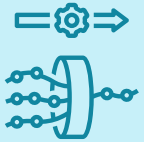
Esto nos ahorra código, y nos permite organizar e identificar mucho mejor las propiedades de cada componente.

```
def __leerPLC():
def __escalar ():
def __validar ():
```

Clase - Analogicas:

Entradas
- Minimo
- Maximo
- Minimo_ing
- Maximo_ing
- Lectura

Salidas
- Lectura escalada



Clase - Presión:

Entradas
- Lectura sensor

Salidas
- Lectura escalada (bar)

Clase - Temperatura:

Entradas
- Lectura sensor

Salidas
- Lectura escalada (°C)

```
def leerTemperatura():
```

```
SensorLaboratorio.leerTemperatura()
```

class polimorfismo:

El polimorfismo permite que objetos de diferentes clases respondan de manera diferente a la misma llamada de un método.

Por ejemplo, un método mover() de una clase “robot” puede no ser el mismo método mover() de una cinta transportadora.

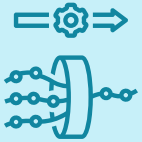
De ésta manera, Python selecciona cuál es el método que le corresponde a cada una de las clases.

```
def __leerPLC():
def __escalar ():
def __validar ():
```

Clase - Analogicas:

Entradas
- Minimo
- Maximo
- Minimo_ing
- Maximo_ing
- Lectura

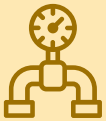
Salidas
- Lectura escalada



Clase - Presión:

Entradas
- Lectura sensor

Salidas
- Lectura escalada (bar)



```
def leerValor():
```

Clase - Temperatura:

Entradas
- Lectura sensor

Salidas
- Lectura escalada (°C)

```
SensorTemp.leerValor()
```

!=

```
SensorPres.leerValor()
```

Ventajas

Con todo lo que hemos visto, podemos afirmar las siguientes ventajas:

La programación de ésta manera me permite crear módulos completamente independientes y cohesivos, de manera muy intuitiva.

Me permite crear bibliotecas de objetos que representen equipos, sensores, actuadores y procesos específicos, y reutilizarlos en otros programas.

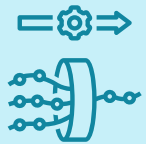
Facilita la división de sistemas complejos en partes más pequeñas y manejables.

Al ser modular, permite encapsular y modelar objetos del mundo real, con sus características y comportamientos reales. Característica clave del gemelo digital.

Al trabajar con jerarquía de objetos, se simplifica la interacción y comunicación entre diferentes componentes de sistemas. Además, permite a cada programador trabajar con objetos específicos sin interferir con un proceso completo.

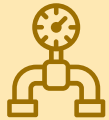
Clase - Analogicas:
Entradas
- Minimo
- Maximo
- Minimo_ing
- Maximo_ing
- Lectura

Salidas
- Lectura escalada



Clase - Presión:
Entradas
- Lectura sensor

Salidas
- Lectura escalada (bar)



Clase - Temperatura:
Entradas
- Lectura sensor

Salidas
- Lectura escalada (°C)

Clase - Motor:
Entradas
- Potencia
- Voltaje

Objeto - MotorCinta:
Entradas
- Potencia: **1/2 HP**
- Voltaje: **220v**
- Fabricante: **Siemens**

Salidas
- Velocidad (RPM): **0**

Objeto - MotorMixer:
Entradas
- Potencia: **5 HP**
- Voltaje: **380v**
- Fabricante: **SEW**

Salidas
- Velocidad (RPM): **1450**

Clases

X

```
02(
    Clases
)
```

class Clase:

¡Es hora de crear nuestras propias clases!

Para definir una clase en Python, se utiliza la palabra clave class, seguida por el nombre de la clase.

```
class Motor:
```

A continuación, definimos los atributos. Tenemos dos tipos de atributos, de **clase** y de **instancia**.

Los atributos de **clase** son variables que se definen dentro de la clase, y son compartidos por todas las instancias. Se definen dentro de la clase, pero fuera de cualquier método.

```
class Motor:
    tipo_motor = "Electrico"
```

Clase - Motor:

Entradas
- Potencia
- Voltaje
- Fabricante

Salidas
- Velocidad (RPM)

class Clase:

Los atributos de instancia son únicos para cada objeto de esa clase. Se definen con el método especial **__init__()** (notar el doble guión bajo)

```
class Motor:
    tipo_motor = "Electrico"
    def __init__(self, potencia, voltaje, fabricante):
        pass #pass es un "placeholder" que no ejecuta ninguna acción
```

Cosas interesantes a notar:

El parámetro **self** es una convención utilizada para referirse al propio objeto sobre el que estamos trabajando. Sirve para pasar parámetros y variables de un método a otro.

No necesariamente debe llamarse “self” (puede llamarse “ObjetoActual”, “Yo”, “me”, etc.), aunque se recomienda llamarlo así para mejor legibilidad.

Clase - Motor:

Entradas
- Potencia
- Voltaje
- Fabricante

Salidas
- Velocidad (RPM)

class Clase:

Dentro de **__init__**, pasaremos aquellos parámetros necesarios y obligatorios para crear el objeto.

Podemos crear otros parámetros no obligatorios, o atributos de salida utilizando self.atributo

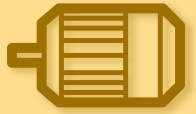
```python
class Motor:
    tipo_motor = "Electrico"
    def __init__(self, potencia, voltaje, fabricante):
        self.potencia = potencia
        self.voltaje = voltaje
        self.fabricante = fabricante

        self.velocidad = 0
        self.caracteristica = "Trifasico"
```

Clase - Motor:

Entradas
- Potencia
- Voltaje
- Fabricante

Salidas
- Velocidad (RPM)



class Clase:

Ahora crearemos dos métodos simples. Más adelante crearemos métodos mucho más complejos para cada uno de los objetos.

```python
class Motor:
    tipo_motor = "Electrico"
    def __init__(self, potencia, voltaje, fabricante):
        self.potencia = potencia
        self.voltaje = voltaje
        self.fabricante = fabricante

        self.velocidad = 0
        self.caracteristica = "Trifasico"

    def arrancar(self):
        print("El motor está en marcha")

    def parar(self):
        print("El motor está detenido")
```

Clase - Motor:

Entradas
- Potencia
- Voltaje
- Fabricante

Salidas
- Velocidad (RPM)

```python
def arrancar():
def parar():
```

class objeto:

Listo! Ya tenemos nuestra clase “motor” creada por completo.

Ahora, solamente tenemos que crear nuestros objetos

```python
class Motor:
    tipo_motor = "Electrico"
    def __init__(self, potencia, voltaje, fabricante):
        self.potencia = potencia
        self.voltaje = voltaje
        self.fabricante = fabricante

        self.velocidad = 0
        self.caracteristica = "Trifasico"

    def arrancar(self):
        print("El motor está en marcha")

    def parar(self):
        print("El motor está detenido")
```

Clase - Motor:

Entradas
- Potencia
- Voltaje
- Fabricante

Salidas
- Velocidad (RPM)

```python
def arrancar():
def parar():
```

class objeto:

Para ello, crearemos una variable y le asignamos la clase motor.

```
motorCinta = Motor()
```

```
---------------------------------------------------------------------------
TypeError Traceback (most recent call last)
c:\Users\Ignacio A. Lavaggi\Desktop\Clase de hoy.ipynb
Cell 13 in 1 ----> 1 motorCinta = Motor()
TypeError: Motor.__init__() missing 3 required positional arguments: 'potencia',
'voltaje', and 'fabricante'
```

Recordemos que tenemos 3 parámetros obligatorios: Potencia, Voltaje y Fabricante.

```
motorCinta = Motor(potencia=0.5,voltaje=220,fabricante="Siemens")
```

Clase - Motor:

Entradas
- Potencia
- Voltaje
- Fabricante

Salidas
- Velocidad (RPM)

```
def arrancar():
def parar():
```

Objeto - MotorCinta:

Entradas
- Potencia: **1/2 HP**
- Voltaje: **220v**
- Fabricante: **Siemens**

Salidas
- Velocidad (RPM): **0**

class objeto:

Listo! Ya tenemos nuestro objeto, podemos crear otro y tenemos dos objetos distintos.

```
motorMixer = Motor(potencia=5,voltaje=380,fabricante="SEW")
```

Con esto, ya podemos acceder a cada uno de los parámetros. Vamos a leer y corregir uno de ellos:

```
motorCinta.caracteristica
```

```
'Trifasico'
```

```
motorCinta.caracteristica = "Monofasico“
motorCinta.caracteristica
```

```
'Monofasico'
```

```
motorMixer.caracteristica
```

```
'Trifasico'
```

Clase - Motor:
Entradas
- Potencia
- Voltaje
- Fabricante

Salidas
- Velocidad (RPM)

```
def arrancar():
def parar():
```

Objeto - MotorCinta:
Entradas
- Potencia: **1/2 HP**
- Voltaje: **220v**
- Fabricante: **Siemens**

Salidas
- Velocidad (RPM): **0**

Objeto - MotorMixer:
Entradas
- Potencia: **5 HP**
- Voltaje: **380v**
- Fabricante: **SEW**

Salidas
- Velocidad (RPM): **1450**

class objeto:

Vamos a crear un objeto más complejo, un servomotor.

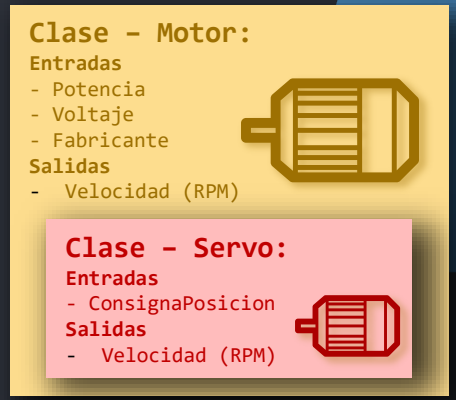
Éste tiene las características de un motor común, pero además tiene otros elementos únicos. Para ello, vamos a heredar algunos atributos de la clase "Motor"

Definimos la clase, indicando a qué familia pertenece.

```
class Servo(Motor):
```

Heredamos las características básicas de "Motor" con la función especial super()

```
class Servo(Motor):
    def __init__(self, potencia, voltaje, fabricante):
        super().__init__(potencia, voltaje, fabricante)
        self.posicion = 0
```



class objeto:

Listo! Ya tenemos nuestra clase, con el atributo adicional “posición”, único de los servomotores.

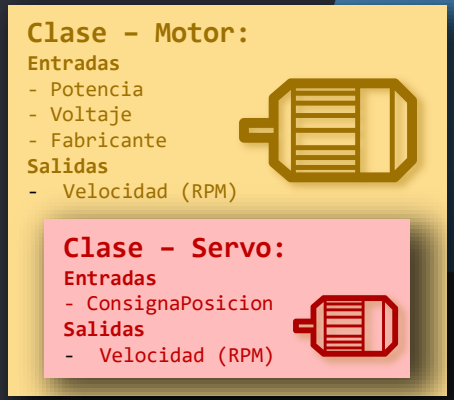
Solamente tenemos que crear nuestro motor y ya está

```
BrazoPosicionador = Servo(potencia=5,voltaje=380,fabricante="SEW")
BrazoPosicionador.arrancar()
```

```
El motor está en marcha
```

```
BrazoPosicionador.posicion = 50
BrazoPosicionador.posicion
```

```
50
```



Python.A_programar!

Ejercicios:

Tanque:

Crear una clase base llamada Tanque con atributos de **instancia** como diámetro, altura y nivel máximo. Además, debe tener otras variables como nivelActual, y caudalConsumo y caudalIngreso.

Definir métodos para el control del tanque: llenar, vaciar, medirLitros

Crea una clase derivada llamada TanqueConValvulas que herede de Tanque e incluya atributos adicionales como el número de válvulas, su estado (abierta/cerrada) y su función (llenado/vaciado)

Define un método vacío llamado abrir_valvula() que represente la acción de abrir una válvula en el tanque con válvulas.