



# P00

Programación Orientada a Objetos

Decorative footer elements consisting of several horizontal bars in white, red, and cyan at the bottom of the slide.

01 {

[Continuación]

- Funcion Main
- Funciones Lambda

}

# Función main {

La **función main** en Python es la función principal de un programa de Python.

Es la función que se ejecuta cuando el programa se inicia y es responsable de iniciar el programa y llamar a otras funciones para realizar el trabajo.

La función main debe tener la siguiente forma:



```
def main():  
    # Coloque su código aquí
```

# Función main {

El cuerpo de la **función main** es donde se escribe el **código principal del programa**. El código en el cuerpo de la **función main** se ejecutará cuando el programa se **inicie**.

La **función main no es obligatoria** en Python. Sin embargo, es una buena práctica tener una **función main** para todos los programas de Python.

}

# Función main {



```
def main():  
    print("Hola, soy el bloque de codigo principal.")  
  
if __name__ == "__main__":  
    main()
```

}

# Función main {



```
# Imports
from funciones import * # Importar TODO

# Funcion main
def main():
    print("Hola, soy el bloque de codigo inicial.")
    saludar()

# Llamar a la funcion main
if __name__ == "__main__":
    main()
```

# Función lambda {

Son **funciones anónimas**, limitadas a una expresión (**una línea**).

Son útiles para crear funciones rápidas que no se necesitan más adelante en su código.



*lambda* parámetros: expresión

}

# Función lambda {

**Parámetros:** Son los parámetros de la función, lo que recibe.

**Expresión:** Es el código que se ejecutará cuando se llame a la función.



*lambda* parámetros: expresión



# Función lambda {



```
multiplicar2 = lambda x, y : x * y
```

```
}
```

# Función lambda {



```
def multiplicar1(x, y):  
    return x * y
```

```
multiplicar2 = lambda x, y : x * y  
print(multiplicar1(2, 4))  
print(multiplicar2(2, 4))
```

# Función lambda {



```
def multiplicarPorFactor(factor):  
    funcionLambda = lambda numero2: factor * numero2  
    return funcionLambda  
  
duplicador = multiplicarPorFactor(2)  
triplicador = multiplicarPorFactor(3)  
print(duplicador(3)) # 6  
print(triplicador(3)) # 9
```

}

# Ejercicio lambda {

Cree una **función lambda** para **sumar dos valores**: x, y.

}

02 {

[ P00 ]

- Conceptos
- Programación Orientada a Objetos
- Clases
- Objetos
- Constructor
- Metodos

}

# Modularidad {

En Python, la modularidad es la **capacidad de dividir un programa grande en módulos más pequeños**. Esto hace que el código sea más fácil de leer, entender y mantener.

Los módulos también pueden ser reutilizados en diferentes programas, lo que ahorra tiempo y esfuerzo.

Para utilizar la modularidad en Python, **se crean módulos que contienen el código que se desea dividir**.

Estos módulos **se pueden importar en otros programas** utilizando la palabra clave **import**. Una vez importados, el código del módulo puede ser utilizado en el programa principal.

}

# Modularidad {

La modularidad también se refiere a la **capacidad de dividir un problema gigante en problemas mucho más pequeños**. Lo cual, nos permitirá tanto resolver el problema de una manera mucho más sencilla, como poder mantenerlo en el tiempo.

Va más allá de segmentar un programa en varios archivos o funciones, pues implica ser cuidadoso a la hora de diseñar nuestra aplicación.

Dependiendo por supuesto del tipo de aplicación que se esté desarrollando.

}

# Abstracción {

La **abstracción de datos** es la técnica para inventar nuevos tipos de datos que sean más adecuados en una aplicación y que, por consiguiente, faciliten la escritura del programa.

Es un concepto que permite ocultar los detalles internos de un objeto y exponer sólo los **detalles esenciales** que son necesarios para interactuar con este.

Esto facilita el uso de los objetos y hace que el código sea más fácil de entender y mantener.

}



# Abstraccion {

Es muy importante recordar siempre que la abstracción es el proceso de enfocarse en los **aspectos importantes** de un objeto y omitir los detalles irrelevantes.

Esto significa que nos concentremos solo en que podemos hacer con él o mejor dicho, cómo podemos aprovechar su implementación sin necesidad de conocer su funcionamiento interno.

Básicamente con la abstracción construimos una **caja negra**.

}

# Tipo de dato abstracto {

En programación, un tipo de dato abstracto (ADT) es un tipo de dato que se define por su comportamiento, en lugar de por su representación interna.

Esto significa que los ADT se pueden utilizar para **representar conceptos del mundo real**.

Los ADT se utilizan a menudo en Programación Orientada a Objetos (OOP) para proporcionar una forma de representar conceptos del mundo real de una manera clara y concisa.

En resumen, un tipo de dato abstracto es un **modelo (estructura) con un número de operaciones**.

}

# Tipo de dato abstracto {

Los tipos abstractos de datos proporcionan numerosos **beneficios** al programador, que se pueden resumir en los siguientes:

1. Permiten una mejor conceptualización y modelización del mundo real.
2. Mejoran la robustez del sistema.
3. Separan la implementación de la especificación.
4. Permiten la extensibilidad del sistema.
5. Permite tener un código: organizado, fácil de entender, reutilizable, eficiente y seguro.

}

# Programación Orientada a Objetos {

La Programación Orientada a Objetos (P00), es un paradigma de programación que permite **organizar nuestro código en base a la idea de que existen “objetos”** y que estos interactúan entre sí, al momento en que creamos nuestros algoritmos.

P00 se basa en la idea de **objetos**. Un objeto es una entidad que tiene datos y comportamiento.

Los **objetos** se utilizan para representar **conceptos del mundo real**, como personas, lugares y cosas.

}

# Programación Orientada a Objetos {

Un **objeto** en Python es una colección de datos (atributos) y comportamientos (métodos), y esto... ¿a que se parece?

Pues se parece a los tipos de datos en Python, tal como los strings que son una cadena de caracteres que a su vez tienen métodos como upper(), lower(), entre otros.

En resumen, **se construyen objetos que almacenan datos y que contienen distintas funcionalidades.**

}

# Paradigma de Programación {

Un paradigma de programación es un **conjunto de conceptos, principios y prácticas** que se utilizan para diseñar y escribir programas de computadora.

Hay muchos paradigmas de programación diferentes, cada uno con sus propias fortalezas y debilidades.

}

# Paradigma de Programación {

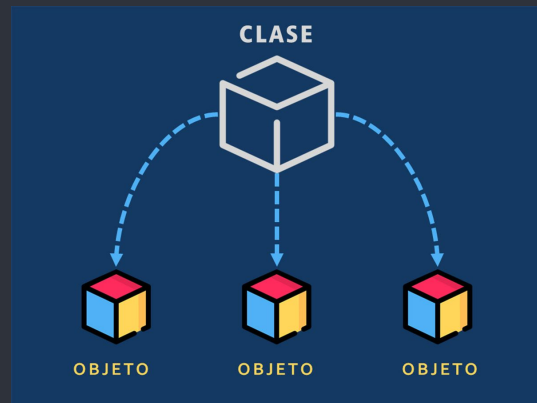
- **Programación imperativa:** este es el paradigma de programación más común. Se basa en la idea de dar **instrucciones al programa paso a paso**.
- **Programación funcional:** Se basa en la idea de funciones. Las funciones son bloques de código que se pueden reutilizar en diferentes partes de un programa.
- **Programación orientada a objetos:** Se basa en la idea de objetos. Los objetos son entidades que tienen datos y comportamiento.
- **Programación lógica:** Se basa en la idea de lógica. Los programas de programación lógica se utilizan para resolver problemas que pueden expresarse como una serie de reglas lógicas.

}

# Clases {

En los lenguajes de programación, tenemos el concepto de **Clase**, que es la forma es como podemos definir un **Tipo de Dato Abstracto**.

En programación orientada a objetos (OOP), una **clase** es un **modelo o plantilla para crear objetos**.





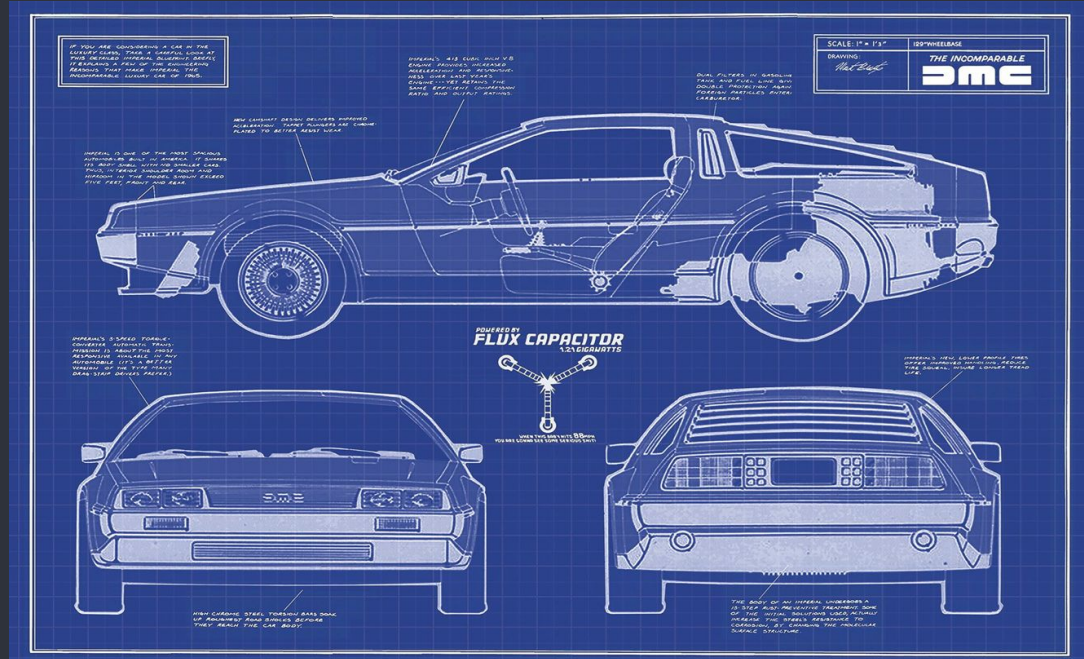
# Clases {

Una clase define los **atributos** y **métodos** de un objeto:

- Los **atributos** son las **características de un objeto**, como: nombre, edad o dirección.
- Los **métodos** son las **acciones que puede realizar un objeto**, como: hablar(), caminar() o comer().

}

# Classes {



# Clases {

En python, una clase se define mediante la palabra clave **class**, colocamos el nombre de la clase seguido de dos puntos:



```
class Carro:
```

}

# Clases. Buenas prácticas {

Algunas de las convenciones para **nombrar clases** en Python son:

- Nombres en **singular**.
- Usar PascalCase.
- Colocar en **mayúscula** todas las letras de una abreviatura. Ejemplo: HTTPServer.

}

# Objetos {

En programación orientada a objetos (OOP), un **objeto** es una entidad que tiene **datos** y **comportamiento**. Los datos de un objeto se almacenan en sus atributos y el comportamiento de un objeto se define en sus métodos.

Los objetos se utilizan para representar conceptos del mundo real, como personas, lugares y cosas.

Por ejemplo, una clase llamada **Persona** podría tener **atributos** como nombre, apellido y edad, y métodos como hablar(), caminar() y comer().

}

# Objetos {



}

# Objetos {

Entonces, una vez que una clase ha sido definida, un programa puede contener una **instancia de dicha clase**, denominada **objeto de la clase**.

**Instanciar** una clase significa crear un objeto a partir de la clase.

Un objeto se crea llamando al **constructor** de la clase.

}

# Objetos {

Aquí hay algunos ejemplos:

- **Una persona:** una persona tiene un nombre, una edad, una dirección y un número de teléfono. También puede hablar, caminar y comer.
- **Un carro:** un carro tiene un motor, un volante, un asiento y un parabrisas. También puede acelerar, girar y frenar.
- **Una casa:** una casa tiene paredes, un techo y una puerta. También puede ser grande o pequeña, y puede tener un jardín o un estacionamiento.

}



# Crear un objeto {

Para crear un objeto debemos igualar una variable (preferible que un nombre tenga relación a la clase) al nombre de la clase seguido de paréntesis.

De esta forma, llamaremos a lo que se conoce como el **constructor de la clase**.



```
carro_1 = Carro()
```

# Partes de una clase {

**Atributos** → Características (variables) de un **objeto**. Los atributos deberían ser identificados con **sustantivos**. Los atributos son variables internas dentro de los objetos.

**Métodos** → Comportamientos (funciones) de un **objeto**. Los métodos deberían ser identificados con **verbos**. Los métodos son funciones que producen algún comportamiento.

**Miembros Estáticos** → Características (atributos) de la **clase**.

}

# Miembros Estáticos {

En programación, los miembros estáticos se **comparten** por todas las instancias de la clase y se pueden acceder a ellos usando el nombre de la clase, en lugar del nombre de una instancia en específico.

Los miembros estáticos se utilizan a menudo para **almacenar datos que son comunes** a todas las instancias de una clase.

En resumen, será un **atributo en común** entre todos los objetos de la clase.

}

# Miembros Estáticos {



```
class Carro:  
    marca = "Ford"
```

}

# Constructor de una clase {

El constructor de una clase es una función especial que se ejecuta cuando se **crea un objeto** de esa clase. El constructor se utiliza para **inicializar los atributos del objeto**.

En Python, el constructor se define con la función **`__init__()`** dentro de la definición de la clase.

Su función es **asignar valores a los objetos** de la clase cuando se crea un objeto.

}

# Constructor de una clase {

Todas las clases tienen una función llamada `__init__()`, que siempre se ejecuta cuando se inicia la clase, es por esto que se le conoce como constructor.

Esta función nos permite asignar valores a las propiedades del objeto u otras operaciones cuando se crea el objeto.

Usaremos el **parámetro `self`** (referencia a la instancia actual de la clase) para poder **acceder a los atributos que pertenecen a la clase**. Podemos llamarlo de cualquier manera, pero para efecto de la clase usaremos `self` o `this`.

}

# Constructor de una clase {

Este es un **método privado**, diferenciado por los dos “\_” que anteceden a su nombre.



```
def __init__(self, modelo, color, anio):  
    self.modelo = modelo  
    self.color = color  
    self.anio = anio
```

}

# Método privado {

En programación, un **método privado** es un método que **no es accesible desde fuera de la clase** en la que se define.

Los métodos privados se utilizan a menudo para ocultar la implementación de una clase y para proteger los datos de la clase de ser accedidos por código no autorizado.

En Python, los métodos privados **se definen precediendo el nombre del método con dos guiones bajos (\_\_)**.

}



# Método privado {



```
def __metodo_privado(self):  
    print('Este es un método privado')
```

}

# Método privado {

```
class Carro2:
    def __init__(self, marca, modelo, año, color, número_de_puertas, número_de_pasajeros):
        self.marca = marca
        self.modelo = modelo
        self.año = año
        self.color = color
        self.número_de_puertas = número_de_puertas
        self.número_de_pasajeros = número_de_pasajeros

    def __metodo_privado(self):
        print('Este es un método privado')
```

}

# Método privado {



```
carro1 = Carro2('Toyota', 'Corolla', 2023, 'rojo', 4, 5)
carro1.__private_method()
# AttributeError: 'Carro2' object has no attribute '__metodo_privado'
```

}

# Constructor de una clase {



```
def __init__(self, modelo, color, anio):  
    self.modelo = modelo  
    self.color = color  
    self.anio = anio
```

```
}
```

# Clase Carro {



*# Nombre de la clase*

**class** Carro:

*# Atributos estaticos*

marca = "Ford"

*# Constructor de la clase*

**def** \_\_init\_\_(self, modelo, color, anio):

self.modelo = modelo

self.color = color

self.anio = anio

}

# Construir objeto {



```
carro_1 = Carro("Fiesta", "Azul", 2011)
```

```
}
```

# Acceder a un atributo {

Para acceder a los atributos del objeto basta con colocar el **nombre de la variable** que lo contiene seguido de un **punto** y luego indicando el **nombre del atributo**.

}

# Acceder a un atributo {



```
carro_1 = Carro("Fiesta", "Azul", 2011)
print(carro_1.modelo) # Atributo
print(carro_1.color) # Atributo
print(carro_1.anio) # Atributo
print(carro_1.marca) # Atributo estatico
```

Fiesta  
Azul  
2011  
Ford

}



# Posición en memoria {



```
carro_1 = Carro("Fiesta", "Azul", 2011)
print(carro_1)
```

```
<Carro.Carro object at 0x000001E87904AF10>
```

}

## Ejercicio 25 {

1. Crea una clase llamada **Persona** que tenga los atributos: nombre, apellido, y edad.
2. Define un **constructor** para la clase **Persona** que inicialice los atributos nombre, apellido, y edad.
3. **Crea una instancia** de la clase **Persona** llamada **persona1** y asigna los valores que desees a los atributos nombre, apellido y edad.
4. **Imprime** el valor de cada atributo de la instancia **persona1**.

}

# Parámetros opcionales {

En Python, los **parámetros opcionales** son **parámetros que se pueden omitir** cuando se llama a una función.

Los **parámetros opcionales** se definen asignando un valor predeterminado al parámetro.

El **valor predeterminado** es el valor que se utilizará si el parámetro no se especifica cuando se llama a la función.

}

# Parámetros opcionales {



```
def __init__(self, modelo, color = "Azul", anio = 2023):  
    self.modelo = modelo  
    self.color = color  
    self.anio = anio
```

```
}
```

# Parámetros opcionales {



```
carro_2 = Carro("Fusion")  
print(carro_2.color) # Azul
```

}

# Métodos {

En programación, un **método** es una función que está asociada con una clase. Los métodos se utilizan para **realizar acciones** en los objetos de una clase.

Se crean tal como hemos creado las funciones anteriormente pero **dentro de una clase** para que sea considerado un método de la misma.

Los métodos se definen usando la palabra **clave def**.

}

# Métodos {



```
def metodo(self):  
    print("Hola soy un metodo de la clase")
```

}

# Métodos {



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

    def metodo(self):
        print("Hola soy un metodo de la clase")
```

}




# Parámetro self {

El **parámetro self** es una referencia a la instancia actual de la clase, y se utiliza para **acceder a los atributos** (variables) de esta.

El **parámetro self** será **siempre** el primer parámetro de cada método de una clase.

}


# Métodos {



```
def imprimir_info(self):  
    print(f"""  
        Nombre: {self.nombre}  
        Apellido: {self.apellido}  
        Edad: {self.edad}  
    """)
```

}

# Métodos {



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

    def mostrar_info(self):
        print(f"""
            Nombre: {self.nombre}
            Apellido: {self.apellido}
            Edad: {self.edad}
        """)
```

# Usar métodos {



```
persona1 = Persona('Juan', 'Pérez', 35)
persona1.metodo()
persona1.mostrar_info()
```

}

# Modificar atributos{



```
persona1 = Persona( 'Juan', 'Pérez', 35)
persona1.edad = 30
print(persona1.edad)
```

30

```
1  Modificar atributos{
```

```
12  
13  }  
14
```



```
def cambiar_edad(self, nueva_edad):  
    self.edad = nueva_edad
```

# Modificar atributos{



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

    def cambiar_edad(self, nueva_edad):
        self.edad = nueva_edad
```

}

# Modificar atributos{



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

    def cambiar_edad(self, nueva_edad):
        self.edad = nueva_edad
```

}



# Modificar atributos{



```
persona1 = Persona('Juan', 'Pérez', 35)
persona1.cambiar_edad(25)
print(25)
```

25

## Ejercicio 26 {

1. Crea una clase llamada Carro que tenga los siguientes atributos: *Marca, Modelo, Año, Color, Número de puertas y Número de pasajeros.*
2. Define un método para la clase Carro llamado **acelerar()** que imprima el mensaje "El carro está acelerando."
3. Define un método para la clase Carro llamado **frenar()** que imprima el mensaje "El carro está frenando."
4. Crea una instancia de la clase Carro llamada **carro1** y asigna los valores **'Toyota', 'Corolla', 2023, rojo, 4, 5** a los atributos marca, modelo, año, color, número de puertas y número de pasajeros.
5. Llama al método **acelerar()** de la instancia **carro1**.
6. Llama al método **frenar()** de la instancia **carro1**.

# Getters. Setters {

En Python, los getters y setters son métodos que se utilizan para acceder y modificar los atributos de una clase.

Los **getters** se utilizan para **obtener el valor de un atributo**, mientras que los **setters** se utilizan para **establecer el valor de un atributo**.

}

# Getters {



```
def getNombre(self):  
    return self.nombre
```

```
def getApellido(self):  
    return self.apellido
```

```
def getEdad(self):  
    return self.edad
```

```
}
```

# Getters{



```
persona1 = Persona('Juan', 'Pérez', 35)
print(persona1.getNombre()) # Juan
```

}

# Setters {



```
def setNombre(self, nuevo_nombre):  
    self.nombre = nuevo_nombre  
  
def setApellido(self, nuevo_apellido):  
    self.apellido = nuevo_apellido  
  
def setEdad(self, nueva_edad):  
    self.edad = nueva_edad
```

# Setters {



```
personal = Persona('Juan', 'Pérez', 35)
personal.setNombre("Pedro")
print(personal.getNombre()) # Pedro
```

}