

**CONQUER
BLOCKS**

PYTHON

**PARADIGMAS DE
PROGRAMACIÓN**

CONQUERBLOCKS



**CODIFICAMOS LA INFORMACION
EN SUCESIONES DE 0s y 1s**



**ESTRUCTURACION, MANIPULACION Y
ALMACENAMIENTO DE DATOS
(Listas, arrays, tuplas, sets y
diccionarios / bucles y condicionales)**



**CODIFICAMOS LA INFORMACION
EN SUCESIONES DE 0s y 1s**

ESTRUCTURAS DE DATOS



**ESTRUCTURACION, MANIPULACION Y
ALMACENAMIENTO DE DATOS**
**(Listas, arrays, tuplas, sets y
diccionarios / bucles y condicionales)**



**CODIFICAMOS LA INFORMACION
EN SUCESIONES DE 0s y 1s**

**LA PROGRAMACION SE BASA EN CREAR UNA
RECETA O SECUENCIA DE INSTRUCCIONES**

ESTRUCTURAS DE DATOS



**ESTRUCTURACION, MANIPULACION Y
ALMACENAMIENTO DE DATOS**
**(Listas, arrays, tuplas, sets y diccionarios /
bucles y condicionales)**

**MANIPULAREMOS LA INFORMACION (LOS
DATOS) PARA OBTENER UN RESULTADO DE LA
MANERA MAS OPTIMA POSIBLE**



CODIFICAMOS LA INFORMACION
EN SUCESIONES DE 0s y 1s

LA PROGRAMACION SE BASA EN CREAR UNA
RECETA O SECUENCIA DE INSTRUCCIONES

ESTRUCTURAS DE DATOS

ESTRUCTURACION, MANIPULACION Y
ALMACENAMIENTO DE DATOS
(Listas, arrays, tuplas, sets y diccionarios /
bucles y condicionales)

ESTRUCTURAS DE PROGRAMACION

MANIPULAREMOS LA INFORMACION (LOS
DATOS) PARA OBTENER UN RESULTADO DE LA
MANERA MAS OPTIMA POSIBLE



PARADIGMAS DE PROGRAMACIÓN

Enfoques o modelos que definen la forma en la que se
deben diseñar, estructurar y escribir los programas



PARADIGMAS DE PROGRAMACIÓN

Programación imperativa

```
mis_numeros = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

sum = 0
for num in mis_numeros:
    if num % 2 == 0:
        sum += num
print(sum)

✓ 0.0s
30
```

Foco: Cómo se deben ejecutar las instrucciones. Estructuración en una secuencia de comandos que modifican el estado del programa y realizan acciones específicas.



PARADIGMAS DE PROGRAMACIÓN

Programación funcional

```
mis_numeros = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
numeros_pares = filter(lambda x: x % 2 == 0, mis_numeros)
suma = sum(numeros_pares)
print(suma)

✓ 0.0s
30
```

Foco: Evaluación de funciones matemáticas. Enfoque en la inmutabilidad y en evitar cambios de estado o sus efectos secundarios.



PARADIGMAS DE PROGRAMACIÓN

Programación estructurada

```

mis_numeros = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

sum = 0
for num in mis_numeros:
    if num % 2 == 0:
        sum += num
print(sum)
✓ 0.0s
30

```

Foco: División en estructuras de control como bucles y decisiones (if statements). Se busca la claridad y la organización del código evitando el uso de saltos no estructurados.



PARADIGMAS DE PROGRAMACIÓN

Programación orientada a objetos (POO)

```

class NumberList:
    def __init__(self, numbers):
        self.numbers = numbers

    def suma_numeros_pares(self):
        sum = 0
        for num in self.numbers:
            if num % 2 == 0:
                sum += num
        return sum

mis_numeros = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
mis_numeros_lista = NumberList(mis_numeros)
print(my_number_list.suma_numeros_pares())
✓ 0.0s
30

```

Foco: Los programas se organizan alrededor de objetos que representan entidades del mundo real. Encapsulamiento de datos y comportamientos en objetos. Conceptos particulares: herencia y polimorfismo.



PARADIGMAS DE PROGRAMACIÓN

Muchos lenguajes de programación admiten múltiples paradigmas y permiten combinarlos según sea necesario.

La elección del paradigma depende del problema a resolver, la preferencia del desarrollador y los requisitos del proyecto.

CONQUER
BLOCKS

PYTHON

INTRODUCCIÓN AL USO DE
FUNCIONES



¿Qué es una función?

Bloques de código a los que asignamos un nombre

Están diseñados para realizar una tarea en específico

Pueden ser usados repetidamente



¿Qué es una función?

```
✓ mi_lista = [1,2,3,4]
  longitud = len(mi_lista)
  print(longitud)

✓ 0.0s

4
```



¿Qué es una función?

```
✓ mi_lista = [1,2,3,4]
  longitud = len(mi_lista)
  print(longitud)

✓ 0.0s
4
```

```
import numpy as np

array_1 = np.array([2,2,2])
array_2 = np.array([1,2,3])

array_suma = np.multiply(array_1, array_2)

print(array_suma)

✓ 0.0s
[2 4 6]
```



¿Qué es una función?

```
✓ mi_lista = [1,2,3,4]
  longitud = len(mi_lista)
  print(longitud)

✓ 0.0s
4
```

```
import numpy as np

array_1 = np.array([2,2,2])
array_2 = np.array([1,2,3])

array_suma = np.multiply(array_1, array_2)

print(array_suma)

✓ 0.0s
[2 4 6]
```



¿Cómo creamos una función?

```
def saludar_usuario():
    """Mostrar un saludo simple
    por pantalla."""
    print("Hola!")
```



¿Cómo creamos una función?

```
def saludar_usuario():
    """Mostrar un saludo simple
    por pantalla."""
    print("Hola!")
```



¿Cómo creamos una función?

The screenshot shows the ConquerBlocks interface with a dark theme. A function definition is being created:

```
def saludar_usuario():
    """Mostrar un saludo simple
    por pantalla."""
    print("Hola!")
```

The code is annotated with a tooltip: "necesario siempre". The output window below shows the result of running the function:

```
saludar_usuario()
✓ 0.0s
Hola!
```



¿Cómo pasamos información a una función?

The screenshot shows the ConquerBlocks interface with a dark theme. A function definition is shown with a parameter:

```
def saludar_usuario(nombre):
    """Mostrar un saludo
    por pantalla."""

    print(f"¡Hola {nombre}!")
```

The code is annotated with a tooltip: "necesario siempre". The output window below shows the result of running the function with the argument "Elena":

```
saludar_usuario("Elena")
✓ 0.0s
¡Hola Elena!
```



¿Cómo pasamos información a una función?

```
def saludar_usuario(nombre):
    """Mostrar un saludo
    por pantalla."""

    print(f"¡Hola {nombre}!")

    saludar_usuario("Elena")
✓ 0.0s

¡Hola Elena!
```



¿Cómo pasamos información a una función?

```
def saludar_usuario(nombre):
    """Mostrar un saludo
    por pantalla."""

    print(f"¡Hola {nombre}!")

    saludar_usuario("Elena")
    saludar_usuario("Maria")
    saludar_usuario("Enrique")
✓ 0.0s

¡Hola Elena!
¡Hola Maria!
¡Hola Enrique!
```



¿Cómo pasamos información a una función?

```
def saludar_usuario(nombre): PARÁMETRO
    """Mostrar un saludo
    por pantalla."""

    print(f"¡Hola {nombre}!")

saludar_usuario("Elena") ARGUMENTO
✓ 0.0s

¡Hola Elena!
```



ARGUMENTOS POSICIONALES

```
def describir_mascota(tipo_animal, nombre_mascota):
    """ Mostrar informacion de mascota."""

    print("Tengo un " + tipo_animal + ".")
    print("Mi " + tipo_animal + " se llama " \
          + nombre_mascota.title() + ".")

describir_mascota("hamster", "harry")

✓ 0.0s

Tengo un hamster.
Mi hamster se llama Harry.
```



ARGUMENTOS POSICIONALES

```
def describir_mascota(tipo_animal, nombre_mascota):
    """ Mostrar informacion de mascota."""

    print("Tengo un " + tipo_animal + ".")
    print("Mi " + tipo_animal + " se llama " \
          + nombre_mascota.title() + ".")

describir_mascota("harry", "hamster")
```

✓ 0.0s

Tengo un harry.
Mi harry se llama Hamster.



ARGUMENTOS DE PALABRA CLAVE

```
def describir_mascota(tipo_animal, nombre_mascota):
    """ Mostrar informacion de mascota."""

    print("Tengo un " + tipo_animal + ".")
    print("Mi " + tipo_animal + " se llama " \
          + nombre_mascota.title() + ".")

describir_mascota(tipo_animal="hamster", nombre_mascota="harry")
```

✓ 0.0s

Tengo un hamster.
Mi hamster se llama Harry.



ARGUMENTOS DE PALABRA CLAVE

```
def describir_mascota(tipo_animal, nombre_mascota):
    """ Mostrar informacion de mascota."""

    print("Tengo un " + tipo_animal + ".")
    print("Mi " + tipo_animal + " se llama " \
          + nombre_mascota.title() + ".")

describir_mascota(nombre_mascota="harry", tipo_animal="hamster")
✓ 0.0s
```

Tengo un hamster.
Mi hamster se llama Harry.



VALORES POR DEFECTO

```
def describir_mascota(nombre_mascota, tipo_animal = "perro"):
    """ Mostrar informacion de mascota."""

    print("Tengo un " + tipo_animal + ".")
    print("Mi " + tipo_animal + " se llama " \
          + nombre_mascota.title() + ".")

describir_mascota(nombre_mascota="Roc")
✓ 0.0s
```

Tengo un perro.
Mi perro se llama Roc.



VALORES POR DEFECTO

```
def describir_mascota(nombre_mascota, tipo_animal = "perro"):  
    """ Mostrar informacion de mascota.  
  
    print("Tengo un " + tipo_animal + ".")  
    print("Mi " + tipo_animal + " se llama " \  
         + nombre_mascota.title() + ".")  
  
describir_mascota(tipo_animal = "hamster", nombre_mascota="harry")  
✓ 0.0s  
  
Tengo un hamster.  
Mi hamster se llama Harry.
```



VALORES POR DEFECTO

```
def describir_mascota(tipo_animal = "perro", nombre_mascota):  
    """ Mostrar informacion de mascota.  
  
    print("Tengo un " + tipo_animal + ".")  
    print("Mi " + tipo_animal + " se llama " \  
         + nombre_mascota.title() + ".")  
  
describir_mascota(nombre_mascota="Roc")  
⊗ 0.0s  
  
Cell In[24], line 1  
def describir_mascota(tipo_animal = "perro", nombre_mascota):  
^  
SyntaxError: non-default argument follows default argument
```



LLAMADAS EQUIVALENTES

```
def describir_mascota(nombre_mascota, tipo_animal = "perro"):
    """ Mostrar informacion de mascota."""

    print("Tengo un " + tipo_animal + ".")
    print("Mi " + tipo_animal + " se llama " +
          nombre_mascota.title() + ".")
```

```
# un perro llamado Roc
describir_mascota(nombre_mascota="roc")
print("----")
describir_mascota("roc")

# un hamster llamado harry
print("=====")
describir_mascota("harry", "hamster")
print("----")
describir_mascota(nombre_mascota = "harry", tipo_animal = "hamster")
print("----")
describir_mascota(tipo_animal = "hamster", nombre_mascota = "harry")
```

```
Tengo un perro.
Mi perro se llama Roc.
-----
Tengo un perro.
Mi perro se llama Roc.
=====
Tengo un hamster.
Mi hamster se llama Harry.
-----
Tengo un hamster.
Mi hamster se llama Harry.
-----
Tengo un hamster.
Mi hamster se llama Harry.
```



ERRORES FRECUENTES

```
def describir_mascota(tipo_animal = "perro", nombre_mascota):
    """ Mostrar informacion de mascota."""

    print("Tengo un " + tipo_animal + ".")
    print("Mi " + tipo_animal + " se llama " +
          nombre_mascota.title() + ".")
```

describir_mascota(nombre_mascota="Roc")

⊗ 0.0s

Cell In[24], line 1

```
def describir_mascota(tipo_animal = "perro", nombre_mascota): ^
```

SyntaxError: non-default argument follows default argument



ERRORES FRECUENTES

```
def describir_mascota(nombre_mascota, tipo_animal = "perro"):
    """ Mostrar informacion de mascota."""

    print("Tengo un " + tipo_animal + ".")
    print("Mi " + tipo_animal + " se llama " \
          + nombre_mascota.title() + ".")

describir_mascota()
⊗ 0.0s

-----
```

TypeError Traceback (most recent call last)
Cell In[31], line 8
 4 print("Tengo un " + tipo_animal + ".")
 5 print("Mi " + tipo_animal + " se llama " \
 6 + nombre_mascota.title() + ".")
--> 8 describir_mascota()

TypeError: describir_mascota() missing 1 required positional argument: 'nombre_mascota'



ERRORES FRECUENTES

```
def describir_mascota(nombre_mascota = "willy", tipo_animal = "perro"):
    """ Mostrar informacion de mascota."""

    print("Tengo un " + tipo_animal + ".")
    print("Mi " + tipo_animal + " se llama " \
          + nombre_mascota.title() + ".")
```

describir_mascota()
✓ 0.0s

Tengo un perro.
Mi perro se llama Willy.

```
def describir_mascota(nombre_mascota, tipo_animal = "perro"):
    """ Mostrar informacion de mascota."""

    print("Tengo un " + tipo_animal + ".")
    print("Mi " + tipo_animal + " se llama " \
          + nombre_mascota.title() + ".")
```

describir_mascota("willy")
✓ 0.0s

Tengo un perro.
Mi perro se llama Willy.



REPASO

- 1. Paradigmas de programación**
- 2. Qué es una función y como crearla**
- 3. Trabajar con parámetros y argumentos**

CONQUER
BLOCKS

CONQUER BLOCKS

PYTHON

FUNCIONES: RETORNO,
ARGUMENTOS ARBITRARIOS Y USO
DE MODULOS

CONQUER BLOCKS



VALORES DE RETORNO

Valores simples:

```
def recibir_nombre_formateado(nombre, apellido):
    """Devuelve el nombre completo bien formateado"""
    nombre_completo = nombre + ' ' + apellido
    return nombre_completo.title()

musico = recibir_nombre_formateado('jimi', 'hendrix')
print(musico)
✓ 0.0s
Jimi Hendrix
```



VALORES DE RETORNO

Valores simples:

```

def recibir_nombre_formateado(nombre, apellido):
    """Devuelve el nombre completo bien formateado"""
    nombre_completo = nombre + ' ' + apellido
    return nombre_completo.title()

musico = recibir_nombre_formateado('jimi', 'hendrix')
print(musico)
✓ 0.0s
Jimi Hendrix

```

4]


```

def recibir_nombre_formateado(primer_nombre, segundo_nombre, apellido):
    """Devuelve el nombre completo bien formateado"""
    nombre_completo = primer_nombre + ' ' + segundo_nombre + ' ' + apellido
    return nombre_completo.title()

musico = recibir_nombre_formateado('jimi', 'hendrix')
print(musico)
⊗ 0.1s

```

5]

```

TypeError                                     Traceback (most recent call last)
Cell In[4], line 6
      3     nombre_completo = primer_nombre + ' ' + segundo_nombre + ' ' + apellido
      4     return nombre_completo.title()
----> 6 musico = recibir_nombre_formateado('jimi', 'hendrix')
      7 print(musico)

TypeError: recibir_nombre_formateado() missing 1 required positional argument: 'apellido'

```

Recordad que podemos añadir argumentos opcionales:



VALORES DE RETORNO

Valores simples:

```

def recibir_nombre_formateado(nombre, apellido):
    """Devuelve el nombre completo bien formateado"""
    nombre_completo = nombre + ' ' + apellido
    return nombre_completo.title()

musico = recibir_nombre_formateado('jimi', 'hendrix')
print(musico)
✓ 0.0s
Jimi Hendrix

```

6]


```

def recibir_nombre_formateado(primer_nombre, segundo_nombre, apellido):
    """Devuelve el nombre completo bien formateado"""
    nombre_completo = primer_nombre + ' ' + segundo_nombre + ' ' + apellido
    return nombre_completo.title()

musico = recibir_nombre_formateado('jimi', "lee", 'hendrix')
print(musico)
✓ 0.0s
Jimi Lee Hendrix

```

7]

Recordad que podemos añadir argumentosopcionales:



VALORES DE RETORNO

Valores simples:

```

def recibir_nombre_formateado(nombre, apellido):
    """Devuelve el nombre completo bien formateado"""
    nombre_completo = nombre + ' ' + apellido
    return nombre_completo.title()

musico = recibir_nombre_formateado('jimi', 'hen')
print(musico)

```

✓ 0.0s
Jimi Hendrix

```

def recibir_nombre_formateado(primer_nombre, apellido, segundo_nombre = ""):
    """Devuelve el nombre completo bien formateado"""
    nombre_completo = primer_nombre + ' ' + segundo_nombre + ' ' + apellido
    return nombre_completo.title()

musico = recibir_nombre_formateado('jimi', 'hendrix')
print(musico)

```

✓ 0.0s
.. Jimi Hendrix

Recordad que podemos añadir argumentos opcionales:



VALORES DE RETORNO

Valores simples:

```

def recibir_nombre_formateado(nombre, apellido):
    """Devuelve el nombre completo bien formateado"""
    nombre_completo = nombre + ' ' + apellido
    return nombre_completo.title()

musico = recibir_nombre_formateado('jimi', 'he')
print(musico)

```

✓ 0.0s
Jimi Hendrix

```

def recibir_nombre_formateado(primer_nombre, apellido, segundo_nombre = ""):
    """Devuelve el nombre completo bien formateado"""
    nombre_completo = primer_nombre + ' ' + segundo_nombre + ' ' + apellido
    return nombre_completo.title()

musico = recibir_nombre_formateado('jimi', 'hendrix', 'lee')
print(musico)

```

✓ 0.0s
Jimi Lee Hendrix

Recordad que podemos añadir argumentosopcionales:



VALORES DE RETORNO

Diccionarios:

```
def construir_persona(nombre, apellido):
    """Devuelve un diccionario con informacion de la persona."""
    persona = {'nombre': nombre, 'apellido': apellido}
    return persona

datos_persona = construir_persona("Juan", "Gomez")
print(datos_persona)

✓ 0.0s

{'nombre': 'Juan', 'apellido': 'Gomez'}
```



VALORES DE RETORNO

Diccionarios:

```
def construir_persona(nombre, apellido, edad = ""):
    """Devuelve un diccionario con informacion de la persona."""
    persona = {'nombre': nombre, 'apellido': apellido}

    if edad:
        persona["edad"] = edad

    return persona

datos_persona = construir_persona("Juan", "Gomez", 25)

print(datos_persona)

✓ 0.0s

{'nombre': 'Juan', 'apellido': 'Gomez', 'edad': 25}
```



VALORES DE RETORNO

Diccionarios:

```
def construir_persona(nombre, apellido, edad = ""):  
    """Devuelve un diccionario con informacion de la persona."""  
    persona = {'nombre': nombre, 'apellido': apellido}  
  
    if edad:  
        persona["edad"] = edad  
  
    return persona  
  
datos_persona = construir_persona("Juan", "Gomez")  
  
print(datos_persona)  
✓ 0.0s  
{'nombre': 'Juan', 'apellido': 'Gomez'}
```



USO DE FUNCIONES EN BUCLES

```
def recibir_nombre_formateado(nombre, apellido):  
    """Devuelve el nombre completo bien formateado"""  
    nombre_completo = nombre + ' ' + apellido  
    return nombre_completo.title()  
  
# Esto es un bucle infinito  
while True:  
    print("\nDime tu nombre completo:")  
    nombre = input("Nombre: ")  
    apellido = input("Apellido: ")  
  
    nombre_formateado = recibir_nombre_formateado(nombre, apellido)  
  
    print("\nHola, " + nombre_formateado + "!")
```

Dime tu nombre completo:

Hola, Juan Gomez!

Dime tu nombre completo:

Hola, Carlos Jimenez!



LISTAS EN FUNCIONES

Pasar una lista:

```
def saludar_usuarios(lista_nombres):
    """Imprime un saludo simple a todos los elementos de la lista."""
    for nombre in lista_nombres:
        mensaje = "Hola, " + nombre.title() + "."
        print(mensaje)

usuarios = ['juan', 'diego', 'lucas']
saludar_usuarios(usuarios)

✓ 0.0s

Hola, Juan.
Hola, Diego.
Hola, Lucas.
```



LISTAS EN FUNCIONES

Modificar una lista:

```
# Lista de diseños que deben ser impresos en 3D
encargos = ['funda iphone', 'robot', 'triangulo']
finalizados = []
# Simulamos que se imprimen los diseños hasta que no
# queda ninguno
# Movemos los diseños impresos a la lista de finalizados
while encargos:
    diseño_actual = encargos.pop()
    # Simulamos la creación de un modelo 3D
    print("Imprimiendo modelo: " + diseño_actual)
    finalizados.append(diseño_actual)
# Mostrar diseños impresos
print("\nLos siguientes modelos han sido impresos:")
for modelo_finalizado in finalizados:
    print(modelo_finalizado)
```

```
Imprimiendo modelo: triangulo
Imprimiendo modelo: robot
Imprimiendo modelo: funda iphone

Los siguientes modelos han sido impresos:
triangulo
robot
funda iphone
```

LISTAS EN FUNCIONES

Modificar una lista:

```
# Lista de diseños que deben ser impresos en 3D
encargos = ['funda iphone', 'robot', 'triangulo']
finalizados = []
# Simulamos que se imprimen los diseños hasta que no
# queda ninguno
# Movemos los diseños impresos a la lista de finalizados
while encargos:
    diseño_actual = encargos.pop()
    # Simular la creación de un modelo 3D
    print("Imprimiendo modelo: " + diseño_actual)
    finalizados.append(diseño_actual)
# Mostrar diseños impresos
print("\nLos siguientes modelos han sido impresos:")
for modelo_finalizado in finalizados:
    print(modelo_finalizado)
```

```
def imprimir_modelos(encargos, finalizados):
    """
    Simula la impresión de cada diseño hasta que todos han sido completados.
    Mueve cada diseño a la lista de finalizados tras imprimirlo.
    """
    while encargos:
        diseño_actual = encargos.pop()

        # Simular la creación de un modelo 3D
        print("Imprimiendo modelo: " + diseño_actual)
        finalizados.append(diseño_actual)

    def muestra_modelos_completados(finalizados):
        """
        Muestra los modelos impresos.
        """
        print("\nLos siguientes modelos han sido impresos:")
        for modelo_finalizado in finalizados:
            print(modelo_finalizado)

    modelos_encargados = ['iphone case', 'robot pendant', 'dodecahedron']
    modelos_completados = []
    imprimir_modelos(modelos_encargados, modelos_completados)
    muestra_modelos_completados(modelos_completados)

✓ 0.0s
Imprimiendo modelo: dodecahedron
Imprimiendo modelo: robot pendant
Imprimiendo modelo: iphone case

Los siguientes modelos han sido impresos:
dodecahedron
robot pendant
iphone case
```



LISTAS EN FUNCIONES

Prevenir la modificación de una lista

```
def imprimir_modelos(encargos, finalizados):
    """
    Simula la impresión de cada diseño hasta que todos han sido completados.
    Mueve cada diseño a la lista de finalizados tras imprimirlo.
    """
    while encargos:
        diseño_actual = encargos.pop()

        # Simular la creación de un modelo 3D
        print("Imprimiendo modelo: " + diseño_actual)
        finalizados.append(diseño_actual)

    def muestra_modelos_completados(finalizados):
        """
        Muestra los modelos impresos.
        """
        print("\nLos siguientes modelos han sido impresos:")
        for modelo_finalizado in finalizados:
            print(modelo_finalizado)

    modelos_encargados = ['iphone case', 'robot pendant', 'dodecahedron']
    modelos_completados = []
    imprimir_modelos(modelos_encargados, modelos_completados)
    muestra_modelos_completados(modelos_completados)

    print("----")
    print(modelos_encargados)
```

```
Imprimiendo modelo: dodecahedron
Imprimiendo modelo: robot pendant
Imprimiendo modelo: iphone case

Los siguientes modelos han sido impresos:
dodecahedron
robot pendant
iphone case
-----
[]
```



LISTAS EN FUNCIONES

Prevenir la modificación de una lista

```

def imprimir_modelos(encargos, finalizados):
    """
        Simula la impresión de cada diseño hasta que todos han sido completados.
        Mueve cada diseño a la lista de finalizados tras imprimirlo.
    """
    while encargos:
        diseño_actual = encargos.pop()
        # Simular la creación de un modelo 3D
        print("Imprimiendo modelo: " + diseño_actual)
        finalizados.append(diseño_actual)

def muestra_modelos_completados(finalizados):
    """
        Muestra los modelos impresos.
    """
    print("\nLos siguientes modelos han sido impresos:")
    for modelo_finalizado in finalizados:
        print(modelo_finalizado)

modelos_encargados = ['iphone case', 'robot pendant', 'dodecahedron']
modelos_completados= []
imprimir_modelos(modelos_encargados[:], modelos_completados)
muestra_modelos_completados(modelos_completados)

print("----")
print(modelos_encargados)

```

```

Imprimiendo modelo: dodecahedron
Imprimiendo modelo: robot pendant
Imprimiendo modelo: iphone case

Los siguientes modelos han sido impresos:
dodecahedron
robot pendant
iphone case
-----
['iphone case', 'robot pendant', 'dodecahedron']

```



USAR UN NUMERO ARBITRARIO DE ARGUMENTOS

Ejemplo: Pedido restaurante

```

def hacer_pizza(*ingredientes):
    """
        Imprimir la lista de ingredientes del pedido
    """
    print(ingredientes)

hacer_pizza("pepperoni")
hacer_pizza("champignons", "pimiento verde", "aceitunas")

✓ 0.0s

('pepperoni',)
('champignons', 'pimiento verde', 'aceitunas')

```

Ingredientes es una tupla



USAR UN NUMERO ARBITRARIO DE ARGUMENTOS

Ejemplo: Pedido restaurante

Ingredientes es una tupla

```
def hacer_pizza(dimension, *ingredientes):
    """Resumen del pedido"""
    print("Has pedido una pizza de", dimension, "cm.")
    print("La pizza contiene los siguientes ingredientes:")
    for ingrediente in ingredientes:
        print("-", ingrediente)
    print("\n")

hacer_pizza(16, "pepperoni")
hacer_pizza(12, "champignons", "pimiento verde", "aceitunas")
```

Has pedido una pizza de 16 cm.
La pizza contiene los siguientes ingredientes:
- pepperoni

Has pedido una pizza de 12 cm.
La pizza contiene los siguientes ingredientes:
- champignons
- pimiento verde
- aceitunas



USAR UN NUMERO ARBITRARIO DE ARGUMENTOS DE PALABRA CLAVE

```
def construir_perfil(nombre, apellido, **informacion_usuario):
    """Construir un diccionario conteniendo todo
    lo que sabemos del usuario"""
    perfil = {}
    perfil[nombre] = nombre
    perfil[apellido] = apellido

    print(informacion_usuario)
    print(type(informacion_usuario))

    for clave, valor in informacion_usuario.items():
        perfil[clave] = valor
    return perfil

perfil_usuario = construir_perfil("alberto", "lopez",
                                    ubicacion="Madrid",
                                    trabajo = "programador")
```

5] ✓ 0.0s

{'ubicacion': 'Madrid', 'trabajo': 'programador'}
<class 'dict'>



USAR UN NUMERO ARBITRARIO DE ARGUMENTOS DE PALABRA CLAVE

```
def construir_perfil(nombre, apellido, **informacion_usuario):
    """Construir un diccionario conteniendo todo
    lo que sabemos del usuario"""
    perfil = {}
    perfil[nombre] = nombre
    perfil[apellido] = apellido

    print(informacion_usuario)
    print(type(informacion_usuario))

    for clave, valor in informacion_usuario.items():
        perfil[clave] = valor
    return perfil

perfil_usuario = construir_perfil("alberto", "lopez",
                                    ubicacion="Madrid",
                                    trabajo = "programador")

✓ 0.0s
{'ubicacion': 'Madrid', 'trabajo': 'programador'}
<class 'dict'>
```



USAR UN NUMERO ARBITRARIO DE ARGUMENTOS DE PALABRA CLAVE

```
def construir_perfil(nombre, apellido, **informacion_usuario):
    """Construir un diccionario conteniendo todo
    lo que sabemos del usuario"""
    perfil = {}
    perfil["nombre"] = nombre
    perfil["apellido"] = apellido

    for clave, valor in informacion_usuario.items():
        perfil[clave] = valor
    return perfil

perfil_usuario = construir_perfil("alberto", "lopez",
                                    ubicacion="Madrid",
                                    trabajo = "programador")
print(perfil_usuario)

✓ 0.0s
{'nombre': 'alberto', 'apellido': 'lopez', 'ubicacion': 'Madrid', 'trabajo': 'programador'}
```



GUARDAR FUNCIONES EN MODULOS

La ventaja de las funciones es la forma en que separan bloques de código de tu programa principal. Al usar nombres descriptivos para tus funciones, tu programa principal será mucho más fácil de seguir.

- ➡ Puedes ir un paso más allá almacenando tus funciones en un archivo separado llamado módulo y luego importando ese módulo en tu programa principal.



GUARDAR FUNCIONES EN MODULOS

pizza.py

```
def hacer_pizza(dimension, *ingredientes):
    """Resumen del pedido"""
    print("Has pedido una pizza de", dimension, "cm.")
    print("La pizza contiene los siguientes ingredientes:")
    for ingrediente in ingredientes:
        print("-", ingrediente)
    print("\n")
```

nombre modulo

import pizza

nombre función

pizza.hacer_pizza(16, 'pepperoni')

pizza.hacer_pizza(12, 'mushrooms', 'green peppers', 'extra cheese')

hacer_pizza.py



GUARDAR FUNCIONES EN MODULOS

pizza.py

```
def hacer_pizza(dimension, *ingredientes):
    """Resumen del pedido"""
    print("Has pedido una pizza de", dimension, "cm.")
    print("La pizza contiene los siguientes ingredientes:")
    for ingrediente in ingredientes:
        print("-", ingrediente)
        print("\n")
```

nombre modulo	import pizza	nombre función	hacer_pizza.py
			hacer_pizza(16, 'pepperoni')
			hacer_pizza(12, 'mushrooms', 'green peppers', 'extra cheese')



GUARDAR FUNCIONES EN MODULOS

pizza.py

```
def hacer_pizza(dimension, *ingredientes):
    """Resumen del pedido"""
    print("Has pedido una pizza de", dimension, "cm.")
    print("La pizza contiene los siguientes ingredientes:")
    for ingrediente in ingredientes:
        print("-", ingrediente)
        print("\n")
```

nombre función

nombre modulo	nombre función	hacer_pizza.py
from pizza import hacer_pizza		
		hacer_pizza(16, 'pepperoni')
		hacer_pizza(12, 'mushrooms', 'green peppers', 'extra cheese')



GUARDAR FUNCIONES EN MODULOS

pizza.py

```
def hacer_pizza(dimension, *ingredientes):
    """Resumen del pedido"""
    print("Has pedido una pizza de", dimension, "cm.")
    print("La pizza contiene los siguientes ingredientes:")
    for ingrediente in ingredientes:
        print("-", ingrediente)
    print("\n")
```

nombre modulo nombre función
from pizza import hacer_pizza as hpi alias
hacer_pizza.py
hpi(16, 'pepperoni')
hpi(12, 'mushrooms', 'green peppers', 'extra cheese')



GUARDAR FUNCIONES EN MODULOS

pizza.py

```
def hacer_pizza(dimension, *ingredientes):
    """Resumen del pedido"""
    print("Has pedido una pizza de", dimension, "cm.")
    print("La pizza contiene los siguientes ingredientes:")
    for ingrediente in ingredientes:
        print("-", ingrediente)
    print("\n")
```

nombre modulo
import pizza as pi alias
hacer_pizza.py
pi.hacer_pizza(16, 'pepperoni')
pi.hacer_pizza(12, 'mushrooms', 'green peppers', 'extra cheese')



GUARDAR FUNCIONES EN MODULOS

pizza.py

```
def hacer_pizza(dimension, *ingredientes):
    """Resumen del pedido"""
    print("Has pedido una pizza de", dimension, "cm.")
    print("La pizza contiene los siguientes ingredientes:")
    for ingrediente in ingredientes:
        print("-", ingrediente)
        print("\n")
```

nombre modulo
from pizza import * todas las funciones
hacer_pizza.py

```
hacer_pizza(16, "pepperoni")
hacer_pizza(12, "champignons", "pimiento verde", "aceitunas")
```



REPASO

- 1. Valores de retorno**
- 2. Funciones en bucles**
- 3. Numero arbitrario de argumentos + argumentos de palabra clave**
- 4. Uso de listas y diccionarios en funciones**
- 5. Uso de modulos**

CONQUER
BLOCKS

CONQUER **BLOCKS**

PYTHON

**FUNCIONES: ESTILO,
RECURSIVIDAD Y MEMOIZACIÓN**

CONQUERBLOCKS



GUIA DE ESTILO

Funciones y módulos deben tener nombres descriptivos
(en minúscula y con “_” separando las palabras)

Las funciones deben incluir un comentario conciso que
explique su función en formato docstring “”” ”””



Sabiendo el nombre de la función, los argumentos necesarios
y los valores de retorno cualquier programador debe poder
integrar la función en sus programas



GUIA DE ESTILO

```
def nombre_funcion(parametro_0, parametro_1='valor por defecto')
```

No debe haber espacios
alrededor del “=”

```
llamada_funcion(parametro_0, parametro_1="valor")
```



GUIA DE ESTILO

PEP 8: <https://peps.python.org/pep-0008/>

Recomienda limitar las líneas de código a 79 caracteres

```
def nombre_funcion(  
    parameter_0, parameter_1, parameter_2,  
    parameter_3, parameter_4, parameter_5):  
    cuerpo de la funcion
```



GUIA DE ESTILO

PEP 8: <https://peps.python.org/pep-0008/>

La separación entre la definición de dos funciones es de dos líneas en blanco

```
def nombre_funcion(  
    parameter_0, parameter_1, parameter_2,  
    parameter_3, parameter_4, parameter_5):  
    cuerpo de la función
```



GUIA DE ESTILO

PEP 8: <https://peps.python.org/pep-0008/>

Los import deben escribirse al comienzo del script después de los comentarios que describen el programa al completo.

```
#-----  
#Name: Union Tool Sample Script  
#Purpose: Runs the Union geoprocessing tool from ArcGIS Pro  
#Author: Esri & Tripp Corbin  
#  
#Created: 09/15/2015  
#Updated: 07/05/2020  
#  
#Usage: Union two feature classes together  
#Software Version: ArcGIS Pro 2.5 Basic  
#-----  
#Import system modules  
import arcpy  
from arcpy import env  
  
# Sets the current workspace to avoid having to specify the full path  
# to the feature classes each time  
arcpy.env.workspace = c:\\student\\\\IntroArcPro\\\\Databases\\\\Trippville.gdb"  
#Runs Union geoprocessing tool on 2 feature classes  
arcpy.Union_analysis(["Parcels", "Floodplains"], "Parcels_Floodplain_Union", "NO_FID",  
0.0003)
```

Información general del script

Importación de módulos

Comentarios en las secciones del script



RECUSIVIDAD



RECUSIVIDAD

Concepto basico en computer science: Se basa en dividir el problema en sub-problemas faciles de resolver uno a uno



RECUSIVIDAD

Se trata de llamar a una función dentro de si misma.

```
def recursividad(i):
    if i ==1:
        return i
    else:
        return recursividad(i-1)

# caso de uso
recursividad(100)
✓ 0.0s
```

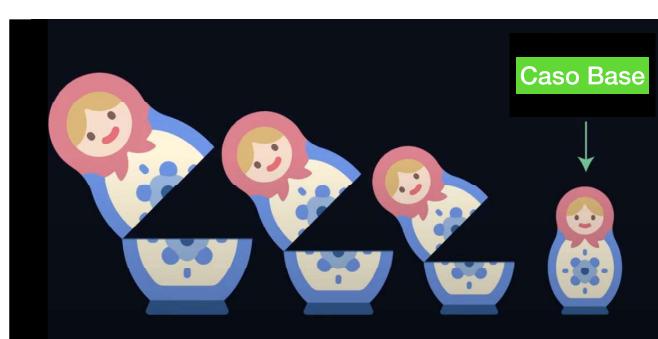


RECUSIVIDAD

Se trata de llamar a una función dentro de si misma.

```
def recursividad(i):
    if i ==1:
        return i
    else:
        return recursividad(i-1)

# caso de uso
recursividad(100)
✓ 0.0s
```





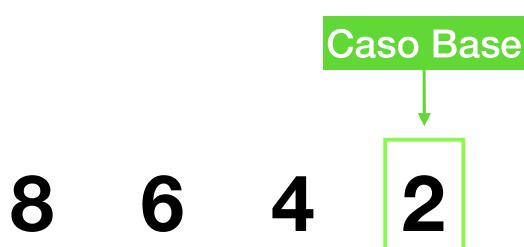
RECUSIVIDAD

EJEMPLO: Escribir todos los números pares positivos menores que 8



RECUSIVIDAD

EJEMPLO: Escribir todos los números pares positivos menores que 8



```
def numeros_pares(num):
    print(num)
    if num == 2:
        return num
    else:
        return numeros_pares(num-2)

numeros_pares(8)
✓ 0.0s
```

8
6
4
2



RECUSIVIDAD

EJEMPLO: Escribir todos los números pares positivos menores que 8

```
def numeros_pares(num):
    for i in range(num, 0, -2):
        print(i)

numeros_pares(8)
✓ 0.0s
```

8
6
4
2

```
def numeros_pares(num):
    print(num)
    if num == 2:
        return num
    else:
        return numeros_pares(num-2)

numeros_pares(8)
✓ 0.0s
```

8
6
4
2

Caso Base



RECUSIVIDAD

EJEMPLO: Serie de Fibonacci

Es una secuencia donde cada numero es la suma de los dos anteriores

0 1 1 2 3 5 8 13 21 ...



RECURSIVIDAD

EJEMPLO: Serie de Fibonacci



RECURSIVIDAD

EJEMPLO: Serie de Fibonacci

Es una secuencia donde cada numero es la suma de los dos anteriores

índice	0	1	2	3	4	5	6	7	8	...
--------	---	---	---	---	---	---	---	---	---	-----

número	0	1	1	2	3	5	8	13	21	...
--------	---	---	---	---	---	---	---	----	----	-----



RECUSIVIDAD

EJEMPLO: Serie de Fibonacci

```
def fibonacci(indice):
    if indice <=1:
        return indice
    else:
        return fibonacci(indice-1)+fibonacci(indice-2)
```



RECUSIVIDAD

EJEMPLO: Serie de Fibonacci

```
def fibonacci(indice):
    if indice <=1:
        return indice
    else:
        return fibonacci(indice-1)+fibonacci(indice-2)

for i in range(0,11):
    print(fibonacci(i))
```

0
1
1
2
3
5
8
13
21
34
55



RECUSIVIDAD

```
def fibonacci(indice):
    if indice <=1:
        return indice
    else:
        return fibonacci(indice-1)+fibonacci(indice-2)
```

$\text{fibonacci}(3) \rightarrow \text{fibonacci}(2) + \text{fibonacci}(1)$



RECUSIVIDAD

```
def fibonacci(indice):
    if indice <=1:
        return indice
    else:
        return fibonacci(indice-1)+fibonacci(indice-2)
```

$\text{fibonacci}(3) \rightarrow \text{fibonacci}(2) + \text{fibonacci}(1)$

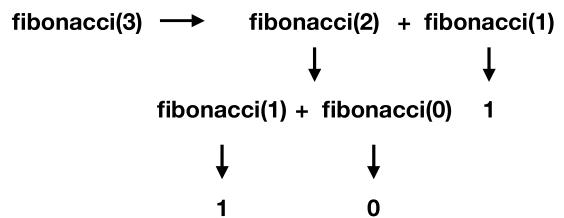


$\text{fibonacci}(1) + \text{fibonacci}(0) \quad 1$



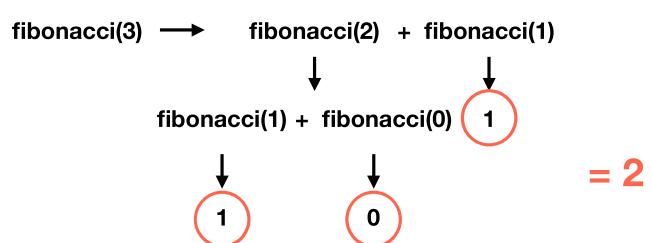
RECUSIVIDAD

```
def fibonacci(indice):
    if indice <=1:
        return indice
    else:
        return fibonacci(indice-1)+fibonacci(indice-2)
```



RECUSIVIDAD

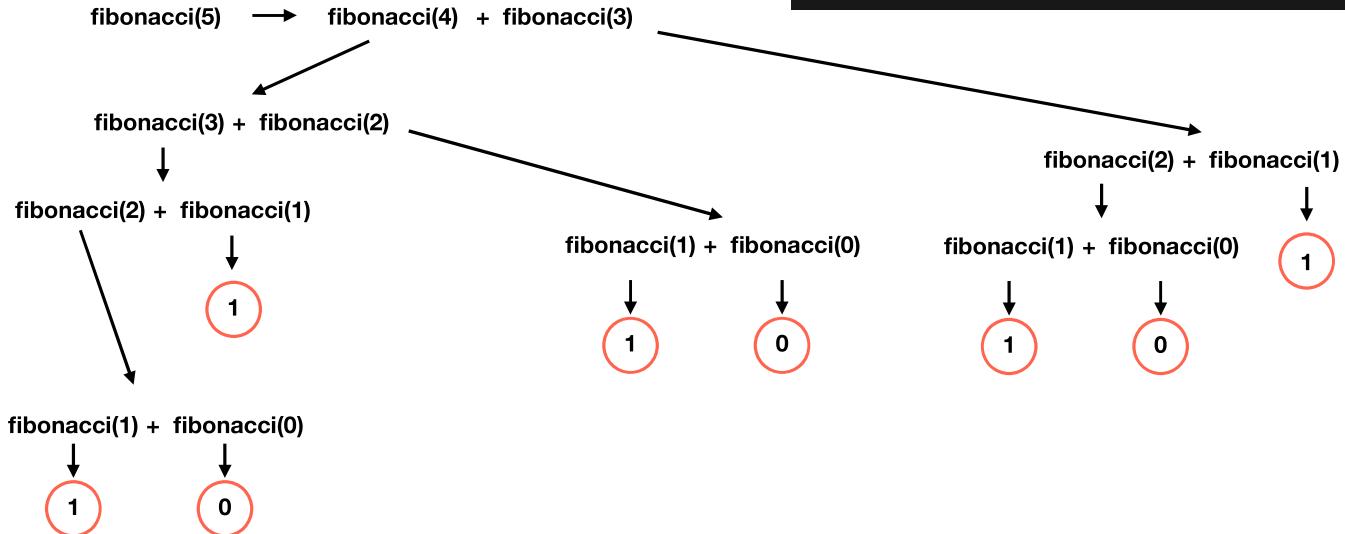
```
def fibonacci(indice):
    if indice <=1:
        return indice
    else:
        return fibonacci(indice-1)+fibonacci(indice-2)
```





RECUSIVIDAD

```
def fibonacci(indice):
    if indice <=1:
        return indice
    else:
        return fibonacci(indice-1)+fibonacci(indice-2)
```



RECUSIVIDAD

EJEMPLO: Serie de Fibonacci

```
def fibonacci(indice):
    if indice <=1:
        return indice
    else:
        return fibonacci(indice-1)+fibonacci(indice-2)

for i in range(0,11):
    print(fibonacci(i))
```

0
1
1
2
3
5
8
13
21
34
55



RECUSIVIDAD

EJEMPLO: Serie de Fibonacci

```
def fibonacci_iter(indice):
    secuencia = [0,1]
    for i in range(indice):
        secuencia.append(secuencia[-1] + secuencia[-2])

    return secuencia[-2]

for i in range(0,11):
    print(fibonacci_iter(i))
```

```
0
1
1
2
3
5
8
13
21
34
55
```



RECUSIVIDAD

EJEMPLO: Serie de Fibonacci

```
start_recursive = time.time()
fibonacci(8)
end_recursive = time.time()
print("total time recursive...", end_recursive - start_recursive)

start_iter = time.time()
fibonacci_iter(8)
end_iter = time.time()
print("total time iterative...", end_iter - start_iter)
```

```
total time recursive... 5.245208740234375e-06
total time iterative... 2.1457672119140625e-06
```



RECUSIVIDAD

EJEMPLO: Serie de Fibonacci

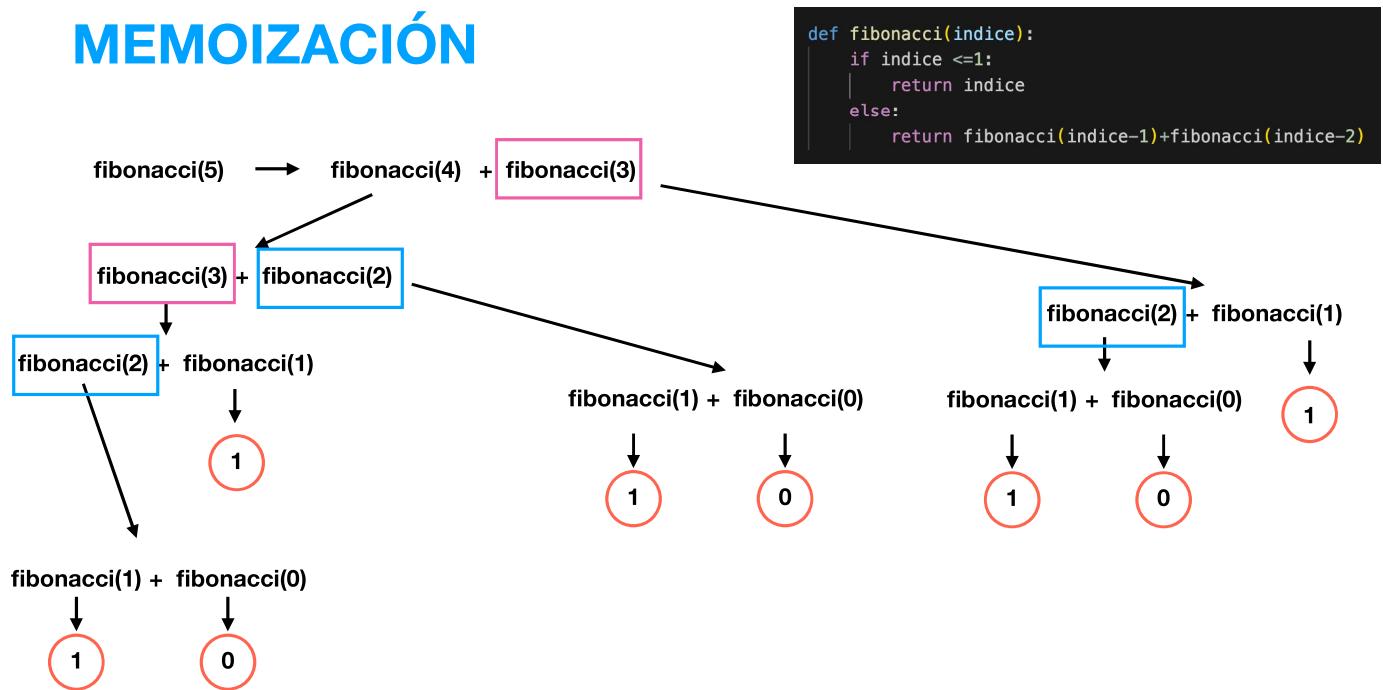
```
start_recursive = time.time()
fibonacci(8)
end_recursive = time.time()
print("total time recursive...", end_recursive - start_recursive)

start_iter = time.time()
fibonacci_iter(8)
end_iter = time.time()
print("total time iterative...", end_iter - start_iter)
```

```
total time recursive... 5.245208740234375e-06
total time iterative... 2.1457672119140625e-06
```



MEMOIZACIÓN





MEMOIZACIÓN

Técnica de optimización en la programación en la cual se almacenan en memoria los resultados de una función para evitar recalcularlos en llamadas futuras con los mismos parámetros.

La memoización puede mejorar significativamente el rendimiento de las funciones que realizan cálculos costosos o repetitivos.



MEMOIZACIÓN - IMPLEMENTACION EXPLICITA

```
fibonacci_cache = {}
def fibonacci_ca(indice):
    #Si tenemos el valor en cache lo devolvemos
    if indice in fibonacci_cache:
        return(fibonacci_cache[indice])

    #Calcular el termino de orden n
    if indice <=1:
        valor = indice
    else:
        valor = fibonacci_ca(indice-1)+fibonacci_ca(indice-2)

    # Guardar el valor en cache y devolverlo
    fibonacci_cache[indice] = valor
    return(valor)
```



MEMOIZACIÓN - IMPLEMENTACION EXPLICITA

```

fibonacci_cache = {}

def fibonacci_ca(indice):
    #Si tenemos el valor en cache lo devolvemos
    if indice in fibonacci_cache:
        return(fibonacci_cache[indice])

    #Calcular el termino de orden n
    if indice <=1:
        valor = indice
    else:
        valor = fibonacci_ca(indice-1)+fibonacci_ca(indice-2)

    # Guardar el valor en cache y devolverlo
    fibonacci_cache[indice] = valor
    return(valor)

```

total time recursive... 5.7220458984375e-06
 total time iterative... 3.0994415283203125e-06
 total time cache... 2.86102294921875e-06



MEMOIZACIÓN - IMPLEMENTACION IMPLICITA

```

from functools import lru_cache

@lru_cache(maxsize = 20)
def fibonacci_cache_implicito(indice):
    #Calcular el termino de orden n
    if indice <=1:
        return indice
    else:
        return fibonacci(indice-1)+fibonacci(indice-2)

def fibonacci_iter(indice):
    secuencia = [0,1]
    for i in range(indice):
        secuencia.append(secuencia[-1] + secuencia[-2])

    return secuencia[-2]

```

total time recursive... 2.47955322265625e-05
 total time iterative... 8.344650268554688e-06
 total time cache explicito... 6.198883056640625e-06
 total time cache implicito... 5.245208740234375e-06

CONQUER
BLOCKS

**CONQUER
BLOCKS**

PYTHON

FUNCIONES LAMBDA Y DECORADORES

CONQUERBLOCKS



FUNCIONES LAMBDA

¿QUÉ SON? SON FUNCIONES ANONIMAS

¿PARA QUÉ SE USAN? PARA ABREVIAR SINTAXIS Y AHORRARNOS TIEMPO

Todo lo que podemos hacer con una función lambda puede hacerse con una función normal, pero no todo lo que podemos hacer con funciones normales puede hacerse con funciones lambda



FUNCIONES LAMBDA

```
def area_triangulo(base, altura):
    return (base*altura/2)

triangulo1 = area_triangulo(5,7)
triangulo2 = area_triangulo(9,6)

print(triangulo1, triangulo2)
```

✓ 0.0s

17.5 27.0



FUNCIONES LAMBDA

```
def area_triangulo(base, altura):
    return (base*altura/2)

triangulo1 = area_triangulo(5,7)
triangulo2 = area_triangulo(9,6)

print(triangulo1, triangulo2)
```

✓ 0.0s

17.5 27.0

```
area_triangulo=lambda base,altura:(base*altura/2)
```

```
triangulo1 = area_triangulo(5,7)
triangulo2 = area_triangulo(9,6)
```

```
print(triangulo1, triangulo2)
```

✓ 0.0s

17.5 27.0



FUNCIONES LAMBDA

The diagram illustrates the equivalence between a standard function definition and a lambda expression. On the left, a standard function `area_triangulo` is defined:

```
def area_triangulo(base, altura):
    return (base*altura/2)
```

This function is then used to calculate the area of two triangles:

```
triangulo1 = area_triangulo(5,7)
triangulo2 = area_triangulo(9,6)

print(triangulo1, triangulo2)
```

The output shows the results: `0.0s`, `17.5`, and `27.0`.

On the right, the same code is shown using a lambda expression:

```
area_triangulo=lambda base,altura:(base*altura/2)

triangulo1 = area_triangulo(5,7)
triangulo2 = area_triangulo(9,6)

print(triangulo1, triangulo2)
```

The output is identical: `0.0s`, `17.5`, and `27.0`.



FUNCIONES LAMBDA

A large blue box at the top contains the text: **NO PUEDE TENER BUCLES O CONDICIONALES. SOLO PUEDE DEVOLVERNOS UN CÁLCULO.**

The code block shows a standard function definition and its use to calculate triangle areas:

```
triangulo1 = area_triangulo(5,7)
triangulo2 = area_triangulo(9,6)

print(triangulo1, triangulo2)
```

The output shows the results: `0.0s`, `17.5`, and `27.0`.

A blue callout box in the center-right of the code area contains the text: **funciones on demand, on the go, online...**



FUNCIONES LAMBDA

EJEMPLO: POTENCIA

```
#al_cubo=lambda numero:pow(numero,3)
al_cubo=lambda numero:numero**3
print(al_cubo(13))

✓ 0.0s

2197
```



FUNCIONES LAMBDA

EJEMPLO: TRABAJO DE FORMATO

```
destacar_valor=lambda comision:"{}! {}".format(comision)
comision_luis = 2300
print(destacar_valor(comision_luis))

✓ 0.0s

i2300! $
```



FILTROS CON FUNCIONES LAMBDA

EJEMPLO: FILTRAR NUMEROS PARES DE UNA LISTA

```
def numero_par(num):
    if num % 2==0:
        return True

numeros=[17,24,7,39,8,51,92]
print(list(filter(numero_par, numeros)))

✓ 0.0s
[24, 8, 92]
```



FILTROS CON FUNCIONES LAMBDA

EJEMPLO: FILTRAR NUMEROS PARES DE UNA LISTA

```
def numero_par(num):
    if num % 2==0:
        return True

numeros=[17,24,7,39,8,51,92]
print(list(filter(lambda numero_par:numero_par%2==0, numeros)))

✓ 0.0s
[24, 8, 92]
```



FUNCIONES LAMBDA COMO CLAVE

EJEMPLO: SORT

```
✓ autores = ["Isaac Asimov", "Frank Herbert", "Douglas Adams"]
autores.sort()
print(autores)
✓ 0.0s
['Douglas Adams', 'Frank Herbert', 'Isaac Asimov']
```



FUNCIONES LAMBDA COMO CLAVE

EJEMPLO: SORT

```
✓ autores = ["Isaac Asimov", "Frank Herbert", "Douglas Adams"]
autores.sort()
print(autores)
✓ 0.0s
['Douglas Adams', 'Frank Herbert', 'Isaac Asimov']
```

```
✓ autores = ["Isaac Asimov", "Frank Herbert", "Douglas Adams"]
autores.sort(key=lambda name:name.split(" ")[-1])
print(autores)
✓ 0.0s
['Douglas Adams', 'Isaac Asimov', 'Frank Herbert']
```



FUNCIONES DECORADORES

SON FUNCIONES QUE AÑADEN FUNCIONALIDADES A FUNCIONES YA EXISTENTES EN NUESTRO PROGRAMA

ESTRUCTURA:

- 3 funciones (A, B, C) donde A recibe como parámetro la función B y devuelve la función C.
- Un decorador devuelve una función



FUNCIONES DECORADORES

SON FUNCIONES QUE AÑADEN FUNCIONALIDADES A FUNCIONES YA EXISTENTES EN NUESTRO PROGRAMA

ESTRUCTURA:

- 3 funciones (A, B, C) donde A recibe como parámetro la función B y devuelve la función C.
- Un decorador devuelve una función

```
        FUNCION A      FUNCION B
def funcion_decorador(funcion):
    def funcion_interna():
        #codigo de funcion interna
        return(funcion_interna)
    return(funcion_interna)

    FUNCION C
```



FUNCIONES DECORADORES

EJEMPLO:

```
def suma():
    print(15+20)

def resta():
    print(10-3)

...
...
...
```



FUNCIONES DECORADORES

EJEMPLO:

```
def suma():
    print(15+20)

def resta():
    print(10-3)

...
...
...
```

```
def funcion_decoradora(funcion_parametro):
    def funcion_interior():
        # Acciones adicionales que decoran
        print("Vamos a realizar el calculo: ")

        funcion_parametro()

        # Acciones adicionales que decoran
        print("Hemos terminado el calculo")

    return(funcion_interior)
```



FUNCIONES DECORADORES

EJEMPLO:

```
def suma():
    print(15+20)

def resta():
    print(10-3)

...
...
...
```

```
def funcion_decoradora(funcion_parametro):
    def funcion_interior():
        # Acciones adicionales que decoran
        print("Vamos a realizar el calculo: ")

        funcion_parametro()

        # Acciones adicionales que decoran
        print("Hemos terminado el calculo")

        return(funcion_interior)

@funcion_decoradora
def suma():
    print(15+20)

@funcion_decoradora
def resta():
    print(10-3)
```

Vamos a realizar el calculo:
35
Hemos terminado el calculo
Vamos a realizar el calculo:
7
Hemos terminado el calculo



FUNCIONES DECORADORES

```
def funcion_decoradora(funcion_parametro):
    def funcion_interior(*args):
        # Acciones adicionales que decoran
        print("Vamos a realizar el calculo: ")

        funcion_parametro(*args)

        # Acciones adicionales que decoran
        print("Hemos terminado el calculo")

    return(funcion_interior)

@funcion_decoradora
def suma(num1, num2, num3):
    print(num1+num2+num3)

@funcion_decoradora
def resta(num1, num2):
    print(num1-num2)

suma(7,5,6)
resta(4,3)
```

Función interna con argumentos

Vamos a realizar el calculo:
18
Hemos terminado el calculo
Vamos a realizar el calculo:
1
Hemos terminado el calculo



FUNCIONES DECORADORES

```

def funcion_decoradora(funcion_parametro):
    def funcion_interior(*args):
        # Acciones adicionales que decoran
        print("Vamos a realizar el calculo: ")

        funcion_parametro(*args)

        # Acciones adicionales que decoran

        print("Hemos terminado el calculo")

    return(funcion_interior)

@funcion_decoradora
def suma(num1, num2, num3):
    print(num1+num2+num3)

@funcion_decoradora
def resta(num1, num2):
    print(num1-num2)

suma(7,5,6)
resta(4,3)

```

Función interna con argumentos

Arguments

```

Vamos a realizar el calculo:
18
Hemos terminado el calculo
Vamos a realizar el calculo:
1
Hemos terminado el calculo

```



FUNCIONES DECORADORES

```

def funcion_decoradora(funcion_parametro):
    def funcion_interior(*args, **kwargs):
        # Acciones adicionales que decoran
        print("Vamos a realizar el calculo: ")

        funcion_parametro(*args, **kwargs)

        # Acciones adicionales que decoran

        print("Hemos terminado el calculo")

    return(funcion_interior)

@funcion_decoradora
def potencia(base, exponente):
    print(base**exponente)

potencia(base=4, exponente=3)
] ✓ 0.0s

```

Función interna con argumentos

```

Vamos a realizar el calculo:
64
Hemos terminado el calculo

```



FUNCIONES DECORADORES

```
def funcion_decoradora(funcion_parametro):
    def funcion_interior(*args, **kwargs):
        # Acciones adicionales que decoran
        print("Vamos a realizar el calculo: ")

        funcion_parametro(*args, **kwargs)

        # Acciones adicionales que decoran

        print("Hemos terminado el calculo")

    return(funcion_interior)

@funcion_decoradora
def potencia(base, exponente):
    print(base**exponente)

potencia(base=4, exponente=3)
```

Función interna con argumentos

Keyword arguments

Vamos a realizar el calculo:
64

Hemos terminado el calculo

**CONQUER
BLOCKS**



PYTHON

USO DE ARCHIVOS

CONQUER **BLOCKS**



OBJETIVO

Aprender a trabajar con archivos y manejar grandes cantidades de datos

- └→ Necesario para analizar y modificar información

Guardar datos para hacer nuestros programas más cómodos para el usuario

- └→ Poder reanudar la ejecución de un script o guardar mensajes de error



LEER UN ARCHIVO

LEER INFORMACIÓN DENTRO DE LA MEMORIA

```

digitos_pi.txt ×  lector_archivo.py
Ejemplo > digitos_pi.txt
1 3.1415926535
2 8979323846
3 2643383279

digitos_pi.txt  lector_archivo.py ×
Ejemplo > lector_archivo.py > archivo_objeto
1 with open('digitos_pi.txt') as archivo_objeto:
2     contenido = archivo_objeto.read()
3     print(contenido)

```

`open()` devuelve un objeto que representa el archivo

Python se encarga de manejar el cierre del archivo



LEER UN ARCHIVO

USAR FILE PATHS - RELATIVO

Linux / OS X

```

with open('path/directorio/digitos_pi.txt') as archivo_objeto:
    contenido = archivo_objeto.read()
    print(contenido)

```

Windows

```

with open('path\directorio\digitos_pi.txt') as archivo_objeto:
    contenido = archivo_objeto.read()
    print(contenido)

```



LEER UN ARCHIVO

USAR FILE PATHS - ABSOLUTO

Linux / OS X

```
file_path = '/home/elena/otros_archivos/archivos_texto/filename.txt'  
with open(file_path ) as archivo_objeto:  
    contenido = archivo_objeto.read()  
    print(contenido)
```

Windows

```
file_path = 'C:\Users\elena\otros_archivos\archivos_texto\filename.txt'  
with open(file_path ) as archivo_objeto:  
    contenido = archivo_objeto.read()  
    print(contenido)
```



LEER UN ARCHIVO

LEER LINEA POR LINEA

```
with open('digitos_pi.txt') as archivo_objeto:  
    for linea in archivo_objeto:  
        print(linea)
```

- (cblocks) MacBook-Pro-5:Ejem
/Ejemplo/lector_archivo.py"
3.1415926535

8979323846

2643383279



LEER UN ARCHIVO

GUARDAR INFORMACION DE UN ARCHIVO

```
(cblOCKS) MacBook-Pro-5:ejemp /Ejemplo/lector_archivo.py"
3.1415926535
8979323846
2643383279

filename = 'digitos_pi.txt'
with open(filename) as archivo_objeto:
    lineas = archivo_objeto.readlines()
    for linea in lineas:
        print(linea.rstrip())
```



LEER UN ARCHIVO

MANEJAR INFORMACION DE UN ARCHIVO

```
(cblOCKS) MacBook-Pro-5:ejemp /Ejemplo/lector_archivo.py"
3.141592653589793238462643383279

filename = 'digitos_pi.txt'      32
with open(filename) as archivo_objeto:
    lineas = archivo_objeto.readlines()
pi_string = ''
for linea in lineas:
    pi_string += linea.strip()
print(pi_string)
print(len(pi_string))
```



ESCRIBIR EN UN ARCHIVO

ARCHIVOS VACIOS

```
filename = "programa.txt"
with open(filename, "w") as archivo_objeto:
    archivo_objeto.write("Estoy aprendiendo python")
```



ESCRIBIR EN UN ARCHIVO

ARCHIVOS VACIOS

```
filename = "programa.txt"
with open(filename, "w") as archivo_objeto:
    archivo_objeto.write("Estoy aprendiendo python")
```



MODOS OPEN

Modo	Descripción
r	Modo de lectura. Abre el archivo para lectura (predeterminado).
w	Modo de escritura. Abre el archivo para escritura, sobrescribe el archivo si existe, o crea uno nuevo si no existe.
a	Modo de anexar. Abre el archivo para escritura, pero agrega contenido al final en lugar de sobrescribirlo. Crea un nuevo archivo si no existe.
x	Modo de creación exclusiva. Abre el archivo para escritura, pero falla si el archivo ya existe.
b	Modo binario. Abre el archivo en modo binario. Se utiliza junto con otros modos, como 'rb' para lectura binaria y 'wb' para escritura binaria.
t	Modo de texto (predeterminado). Abre el archivo en modo de texto para lectura o escritura. Se utiliza junto con otros modos, como 'rt' para lectura de texto y 'wt' para escritura de texto.



MODOS OPEN

r+	Modo de actualización para lectura y escritura. El puntero se coloca al principio del archivo.
w+	Modo de actualización para lectura y escritura. El archivo se sobrescribe o crea uno nuevo.
a+	Modo de actualización para lectura y escritura. El puntero se coloca al final del archivo.



ESCRIBIR EN UN ARCHIVO

ARCHIVOS VACIOS

```
filename = "programa.txt"
with open(filename, "w") as archivo_objeto:
    archivo_objeto.write("Estoy aprendiendo python.")
    archivo_objeto.write("Estoy en el modulo avanzado.")
```

≡ programa.txt ×

Ejemplo > ≡ programa.txt

1 Estoy aprendiendo python. Estoy en el modulo avanzado.



ESCRIBIR EN UN ARCHIVO

ARCHIVOS VACIOS

```
1 filename = "programa.txt"
2 with open(filename, "w") as archivo_objeto:
3     archivo_objeto.write("Estoy aprendiendo python. \n")
4     archivo_objeto.write("Estoy en el modulo avanzado.")
```

≡ programa.txt ×

Ejemplo > ≡ programa.txt

1 Estoy aprendiendo python.
2 Estoy en el modulo avanzado.



AÑADIR ELEMENTOS A UN ARCHIVO

ARCHIVOS VACIOS

```
filename = "programa.txt"
with open(filename, "w") as archivo_objeto:
    archivo_objeto.write("Estoy aprendiendo python. \n")
    archivo_objeto.write("Estoy en el modulo avanzado.\n")

with open(filename, 'a') as file_object:
    file_object.write("Estoy creando un nuevo set de datos.\n")
    file_object.write("Y separo las lineas correctamente.\n")
```

```
programa.txt X
templo > programa.txt
1 Estoy aprendiendo python.
2 Estoy en el modulo avanzado.
3 Estoy creando un nuevo set de datos.
4 Y separo las lineas correctamente.
5
```



AÑADIR ELEMENTOS A UN ARCHIVO

ARCHIVOS VACIOS

```
filename = "programa.txt"
with open(filename, "w") as archivo_objeto:
    archivo_objeto.write("Estoy aprendiendo python. \n")
    archivo_objeto.write("Estoy en el modulo avanzado.\n")

with open(filename, 'a') as file_object:
    file_object.write("Estoy creando un nuevo set de datos.\n")
    file_object.write("Y separo las lineas correctamente.\n")

f = open(nombre_archivo, "w+")
f.write(contenido en forma de string)
```

```
programa.txt X
templo > programa.txt
1 Estoy aprendiendo python.
2 Estoy en el modulo avanzado.
3 Estoy creando un nuevo set de datos.
4 Y separo las lineas correctamente.
5
```



MANEJAR ARCHIVOS CON NUMPY

LECTURA:

```
import numpy as np

# Leer datos de un archivo CSV
data = np.loadtxt('datos.csv', delimiter=',')

# Leer datos de un archivo de texto
data = np.loadtxt('datos.txt')

# Leer datos de un archivo con encabezados
data = np.genfromtxt('datos.csv', delimiter=',', skip_header=1)
```



MANEJAR ARCHIVOS CON NUMPY

ESCRITURA:

```
import numpy as np

# Crear datos de ejemplo
data = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]])

# Guardar datos en un archivo CSV
np.savetxt('datos.csv', data, delimiter=',')

# Guardar datos en un archivo de texto
np.savetxt('datos.txt', data)

# Guardar datos en un archivo con encabezados
header = 'Columna 1, Columna 2, Columna 3'
np.savetxt('datos.csv', data, delimiter=',', header=header, comments='')
```



MANEJAR ARCHIVOS CON NUMPY

ESCRITURA:

```
import numpy as np

# Crear datos de ejemplo
data = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]])

# Guardar datos en un archivo CSV
np.savetxt('datos.csv', data, delimiter=',')

# Guardar datos en un archivo de texto
np.savetxt('datos.txt', data)

# Guardar datos en un archivo con encabezados
header = 'Columna 1, Columna 2, Columna 3'
np.savetxt('datos.csv', data, delimiter=',', header=header, comments='')
```



TRABAJAR CON ARCHIVOS JSON

En muchos programas querremos guardar el input de los usuarios.
(p.e. preferencias en un juego o datos de visualización)

Guardaremos la información en estructuras de datos como listas o diccionarios.

Cuando se cierre el programa querremos que la información de los settings no se pierda. Para ello podremos guardar la información en archivos (por ejemplo de tipo json)



TRABAJAR CON ARCHIVOS JSON

JSON = JavaScript Object Notation

Desarrollado originalmente para JavaScript.

Es uno de los formatos más usados en muchos lenguajes de programación, también python.



TRABAJAR CON ARCHIVOS JSON

```
import json

numeros = [2, 3, 5, 7, 11, 13]
filename = 'numeros.json'
with open(filename, 'w') as f_obj:
    json.dump(numeros, f_obj)
```



TRABAJAR CON ARCHIVOS JSON

```
import json

numeros = [2, 3, 5, 7, 11, 13]
filename = 'numeros.json'
with open(filename, 'w') as f_obj:
    json.dump(numeros, f_obj)
```

Datos Archivo

Convierte un objeto de python en un json string

```
{} numeros.json X  ≡ programa.txt
Ejemplo > {} numeros.json > ...
1   [2, 3, 5, 7, 11, 13]
```



TRABAJAR CON ARCHIVOS JSON

```
import json
filename = 'numeros.json'
with open(filename) as f_obj:
    numeros = json.load(f_obj)
    print(numeros)
```



TRABAJAR CON ARCHIVOS JSON

```
import json  
filename = 'numeros.json'  
with open(filename) as f_obj:  
    numeros = json.load(f_obj)  
    print(numeros)
```

```
(cblocks) MacBook-Pro-5:Ej  
/Ejemplo/json_files.py"  
[2, 3, 5, 7, 11, 13]  
<class 'list'>
```

→ Abrimos el archivo en reading mode

Convierte json string en un objeto de python

CONQUER
BLOCKS



PYTHON

EXCEPCIONES



OBJETIVO

- Manejar errores para que nuestros programas no tengan un crash cuando se encuentren con situaciones inesperadas.

Objetos de tipo excepción

- Manejar excepciones en distintos casos de uso:

Errores aritméticos
Ausencia de archivos



Añadir solidez y estabilidad a nuestros códigos



ZeroDivisionError

```
✓ print(5/0)
  ⏺ 0.1s
-----
ZeroDivisionError
Cell In[1], line 1
----> 1 print(5/0)

ZeroDivisionError: division by zero
```

Un numero no puede dividirse por 0

Objeto de excepción

Se crea cuando python no
puede realizar aquello que
se le pide



ZeroDivisionError

```
✓ print(5/0)
  ⏺ 0.1s
-----
ZeroDivisionError
Cell In[1], line 1
----> 1 print(5/0)

ZeroDivisionError: division by zero
```

Nuestro objetivo será indicarle a python
que hacer cuando ocurra una excepción
de este tipo



Uso de bloques try-except

```
try:  
    print(5/0)  
except ZeroDivisionError:  
    print("¡No puedes dividir por cero!")  
  
✓ 0.0s  
  
¡No puedes dividir por cero!
```



Uso de bloques try-except

```
try:  
    print(5/0)  
except ZeroDivisionError:  
    print("¡No puedes dividir por cero!")  
  
✓ 0.0s  
  
¡No puedes dividir por cero!
```

Si funciona, python ignorará el bloque en el except.



Uso de bloques try-except

```
try:  
    print(5/0)  
except ZeroDivisionError:  
    print("¡No puedes dividir por cero!")  
  
✓ 0.0s  
  
¡No puedes dividir por cero!
```

Si no funciona, python buscará un bloque `except` que se corresponda con el error que ha producido



Uso de bloques try-except

```
try:  
    print(5/0)  
except ZeroDivisionError:  
    print("¡No puedes dividir por cero!")  
  
✓ 0.0s  
  
¡No puedes dividir por cero!
```

Si no funciona, python buscará un bloque `except` que se corresponda con el error que ha producido

↓
ZeroDivisionError



Uso de bloques try-except

ZeroDivisionError

```
try:  
    print(5/0)  
except ZeroDivisionError:  
    print("¡No puedes dividir por cero!")  
  
✓ 0.0s  
  
¡No puedes dividir por cero!
```

Mensaje de error personalizado

+

El código seguirá ejecutándose (no habrá crash) porque le hemos indicado a python como manejar el error.



Evitar crashes

```
print("Dame dos numeros y para dividir.")  
print("Introduce 's' para salir.")  
  
while True:  
    numero1 = input("\nPrimer numero: ")  
    if numero1 == 's':  
        break  
    numero2 = input("Segundo numero: ")  
    if numero2 == 's':  
        break  
    resultado = int(numero1) / int(numero2)  
    print(resultado)
```



Evitar crashes

```
print("Dame dos numeros y para dividir.")
print("Introduce 's' para salir.")
```

```
while True:
    numero1 = input("\nPrimer numero: ")
    if numero1 == 's':
        break
    numero2 = input("Segundo numero: ")
    if numero2 == 's':
        break
    resultado = int(numero1) / int(numero2)
    print(resultado)
```

```
ZeroDivisionError
Cell In[8], line 11
    resultado = int(numero1) / int(numero2)
    ^
--> 11 resultado = int(numero1) / int(numero2)
    12 print(resultado)

zeroDivisionError: division by zero
```



Evitar crashes

```
print("Dame dos numeros y para dividir.")
print("Introduce 's' para salir.")

while True:
    numero1 = input("\nPrimer numero: ")
    if numero1 == 's':
        break
    numero2 = input("Segundo numero: ")
    if numero2 == 's':
        break
    try:
        resultado = int(numero1) / int(numero2)

    except ZeroDivisionError:
        print("You can't divide by 0!")

    else:
        print(resultado)
```

```
Dame dos numeros y para dividir.
Introduce 's' para salir.

Primer numero: 3
Segundo numero: 4
0.75

Primer numero: 5
Segundo numero: 0
You can't divide by 0!

Primer numero: 6
Segundo numero: 7
0.8571428571428571
```



Evitar crashes

```
print("Dame dos numeros y para dividir.")
print("Introduce 's' para salir.")

while True:
    numero1 = input("\nPrimer numero: ")
    if numero1 == 's':
        break
    numero2 = input("Segundo numero: ")
    if numero2 == 's':
        break
    try:
        resultado = int(numero1) / int(numero2)

    except ZeroDivisionError:
        print("You can't divide by 0!")

    else:
        print(resultado)
```

Dame dos numeros y para dividir.
Introduce 's' para salir.

Primer numero: 3
Segundo numero: 4
0.75

Primer numero: 5
Segundo numero: 0
You can't divide by 0!

Primer numero: 6
Segundo numero: 7
0.8571428571428571



FileNotFoundException

```
filename = "test.txt"

with open(filename) as f_obj:
    contenido = f_obj.read()
```



FileNotFoundException

```
with open(filename) as f_obj:  
FileNotFoundException [Errno 2] No such file or directory: 'test.txt'  
  
filename = "test.txt"  
  
with open(filename) as f_obj:  
    contenido = f_obj.read()
```



FileNotFoundException

```
filename = "test.txt"  
try:  
    with open(filename) as f_obj:  
        contents = f_obj.read()  
except FileNotFoundError:  
    msj = "Lo siento, el archivo " + filename + " no existe."  
    print(msj)
```



FileNotFoundException

```
filename = "test.txt"
try:
    with open(filename) as f_obj:
        contents = f_obj.read()
except FileNotFoundError:
    msj = "Lo siento, el archivo " + filename + " no existe."
    print(msj)
```

```
(cblocks) MacBook-Pro-5:Clase 8 Elena$ python3 ./vanzado/Clase 8/Ejemplo/FileNotFounds_test.py
Lo siento, el archivo test.txt no existe.
(cblocks) MacBook-Pro-5:Clase 8 Elena$
```



Ejemplo Practico

ANALISIS DE TEXTOS

→ **Contar el numero de palabras en un texto**

```
filename = "test.txt"
try:
    with open(filename) as f_obj:
        contenido = f_obj.read()
except FileNotFoundError:
    msj = "Lo siento, el archivo " + filename + " no existe."
    print(msj)

else:
    # Count the approximate number of words in the file.
    palabras = contenido.split()
    num_palabras = len(palabras)
    print("El archivo " + filename + " tiene " + str(num_palabras) + " palabras.")
```

```
(cblocks) MacBook-Pro-5:Clase 8/Ejemplo/analisis_de_texto.py
vanzado/Clase 8/Ejemplo/analisis_de_texto.py
El archivo test.txt tiene 125 palabras.
(cblocks) MacBook-Pro-5:Clase 8/Ejemplo/Elena$
```



Ejemplo Practico

(<http://gutenberg.org/>)

ANALISIS DE TEXTOS → **Trabajar con multiples archivos**

```
def contar_palabras(filename):
    """Count the approximate number of words in a file."""
    try:
        with open(filename) as f_obj:
            contenido = f_obj.read()
    except FileNotFoundError:
        msj = "Lo siento, el archivo " + filename + " no existe."
        print(msj)
    else:
        # Count approximate number of words in the file.
        palabras = contenido.split()
        num_palabras = len(palabras)
        print("El archivo " + filename + " tiene " + str(num_palabras) +
              " palabras.")

filenames = ['alice.txt', 'siddhartha.txt', 'moby_dick.txt', 'metamorfosis.txt']

for filename in filenames:
    contar_palabras(filename)
```

vanzado/Clase 8/Ejemplo/analisis_de_texto_2.py"
 El archivo alice.txt tiene 125 palabras.
 El archivo siddhartha.txt tiene 267 palabras.
 Lo siento, el archivo moby_dick.txt no existe.
 El archivo metamorfosis.txt tiene 192 palabras.



Errores silenciosos

```
def contar_palabras(filename):
    """Count the approximate number of words in a file."""
    try:
        with open(filename) as f_obj:
            contenido = f_obj.read()
    except FileNotFoundError:
        pass
    else:
        # Count approximate number of words in the file.
        palabras = contenido.split()
        num_palabras = len(palabras)
        print("El archivo " + filename + " tiene " + str(num_palabras) +
              " palabras.")

filenames = ['alice.txt', 'siddhartha.txt', 'moby_dick.txt', 'metamorfosis.txt']

for filename in filenames:
    contar_palabras(filename)
```



Errores silenciosos

```
def contar_palabras(filename):
    """Count the approximate number of words in a file."""
    try:
        with open(filename) as f_obj:
            contenido = f_obj.read()
    except FileNotFoundError:
        pass ←
    else:
        # Count approximate number of words in the file.
        palabras = contenido.split()
        num_palabras = len(palabras)
        print("El archivo " + filename + " tiene " + str(num_palabras) +
              " palabras.")

filenames = ['alice.txt', 'siddhartha.txt', 'moby_dick.txt', 'metamorfosis.txt']

for filename in filenames:
    contar_palabras(filename)
```



Errores silenciosos

```
def contar_palabras(filename):
    """Count the approximate number of words in a file."""
    try:
        with open(filename) as f_obj:
            contenido = f_obj.read()
    except FileNotFoundError:
        pass ←
    else:
        # Count approximate number of words in the file.
        palabras = contenido.split()
        num_palabras = len(palabras)
        print("El archivo " + filename + " tiene " + str(num_palabras) +
              " palabras.")

filenames = ['alice.txt', 'siddhartha.txt', 'moby_dick.txt', 'metamorfosis.txt']

for filename in filenames:
    contar_palabras(filename)
```

vanzado/Clase 8/Ejemplo/analisis_de_texto_2.py"
El archivo alice.txt tiene 125 palabras.
El archivo siddhartha.txt tiene 267 palabras.
El archivo metamorfosis.txt tiene 192 palabras.



¿Qué errores debemos reportar?

En el caso anterior...

- Si el usuario sabe que textos deben ser analizados apreciará saber por qué algunos no han podido ser analizados.
- Si en cambio el usuario espera resultados pero no sabe que textos deben analizarse puede que no necesite saber que hay algunos que no están disponibles.



¿Qué errores debemos reportar?

- Dar al usuario información que no necesita puede ser contraproducente y reducir la utilidad de tu programa.
- Un código bien escrito es proclive a muy pocos errores lógicos o de sintaxis

Posibles focos de error:

Siempre que tu programa dependa de información externa...

input del usuario
existencia de un archivo
disponibilidad de red de conexión

...

CONQUER
BLOCKS

**CONQUER
BLOCKS**

PYTHON

PROGRAMACION ORIENTEADA
A OBJETOS (POO)

CONQUERBLOCKS



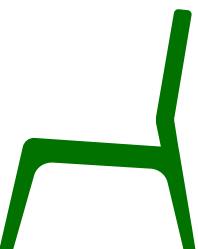
OBJETOS Y CLASES

CLASES →

Representaciones del mundo
real

Ejemplo:

SILLA





OBJETOS Y CLASES

CLASES →

Representaciones del mundo real

Ejemplo:

COCHE



OBJETOS Y CLASES

OBJETOS →

Una instancia de esa clase

Ejemplo:

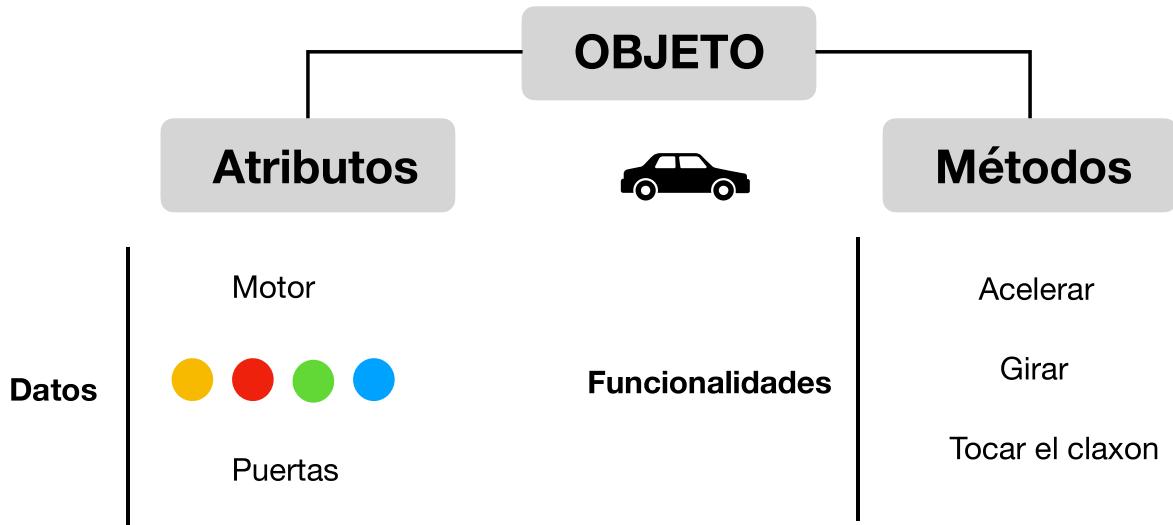
**MI
COCHE**



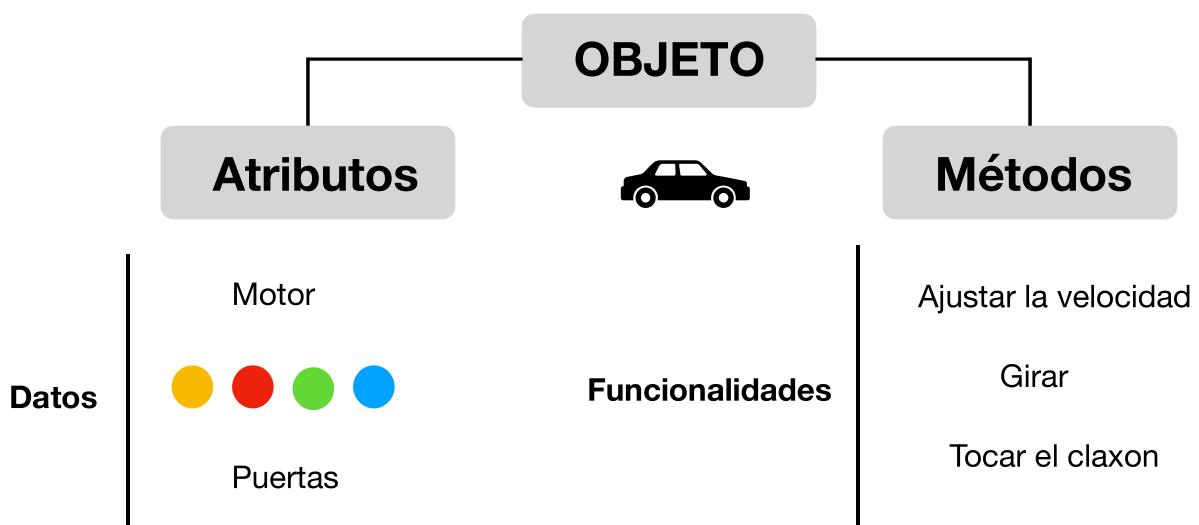
FERRARI



OBJETOS Y CLASES



OBJETOS Y CLASES





OBJETOS Y CLASES

EJEMPLO:

```
class Fruta:  
    def __init__(self):  
        self.nombre = "manzana"  
        self.color = "rojo"
```



OBJETOS Y CLASES

EJEMPLO:

```
class Fruta:  
    def __init__(self):  
        self.nombre = "manzana"  
        self.color = "rojo"
```



OBJETOS Y CLASES

EJEMPLO:

```
        nombre
class Fruta:
    def __init__(self):
        self.nombre = "manzana"
        self.color = "rojo"
```



OBJETOS Y CLASES

EJEMPLO:

```
class Fruta:
    def __init__(self): función de inicialización
        self.nombre = "manzana"
        self.color = "rojo"
```



OBJETOS Y CLASES

EJEMPLO:

```
class Fruta:  
    def __init__(self): función de  
        self.nombre = "manzana"  
        self.color = "rojo"
```

Atributos



OBJETOS Y CLASES

EJEMPLO:

```
class Fruta:  
    def __init__(self):  
        self.nombre = "manzana"  
        self.color = "rojo"  
  
mi_fruta = Fruta()
```

PODEMOS ASIGNAR
NUESTRA CLASE A
UNA VARIABLE



OBJETOS Y CLASES

EJEMPLO:

```
class Fruta:  
    def __init__(s  
        self.nombre  
        self.color
```

```
mi_fruta = Fruta()
```

```
print(mi_fruta.color)  
print(mi_fruta.nombre)
```

✓ 0.0s

rojo
manzana

PODEMOS ACCEDER
A LOS ATRIBUTOS



OBJETOS Y CLASES

EJEMPLO:

```
class Fruta:  
    def __init__(s  
        self.nombre  
        self.color  
  
mi_fruta = Fruta()
```

```
mi_fruta.color = "verde"  
mi_fruta.nombre = "pera"  
print(mi_fruta.color)  
print(mi_fruta.nombre)
```

✓ 0.0s

verde
pera

REASIGNAR VALOR
DE ATRIBUTOS



OBJETOS Y CLASES

EJEMPLO:

```
class Fruta:  
    def __init__(self, nombre, color):  
        self.nombre = nombre  
        self.color = color  
  
mi_fruta = Fruta("manzana", "rojo")
```

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

```
mi_fruta = Fruta("manzana", "rojo")  
print(mi_fruta.color)  
print(mi_fruta.nombre)
```

✓ 0.0s
rojo
manzana



OBJETOS Y CLASES

EJEMPLO:

```
class Fruta:  
    def __init__(self, nombre, color):  
        self.nombre = nombre  
        self.color = color  
  
mi_fruta = Fruta("manzana", "rojo")
```

Parámetros

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

```
mi_fruta = Fruta("manzana", "rojo")  
print(mi_fruta.color)  
print(mi_fruta.nombre)
```

✓ 0.0s
rojo
manzana



OBJETOS Y CLASES

EJEMPLO:

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

mi_fruta = Fruta("manzana", "rojo")
```

Parámetros

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

Asignamos los parámetros a los atributos
`mi_fruta = Fruta("manzana", "rojo")`
`print(mi_fruta.color)`
`print(mi_fruta.nombre)`

✓ 0.0s
 rojo
 manzana



OBJETOS Y CLASES

EJEMPLO:

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

mi_fruta = Fruta("manzana", "rojo")
```

Argumentos

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

`mi_fruta = Fruta("manzana", "rojo")`
`print(mi_fruta.color)`
`print(mi_fruta.nombre)`

✓ 0.0s
 rojo
 manzana



OBJETOS Y CLASES

EJEMPLO:

```
manzana = Fruta("manzana", "rojo")
platano = Fruta("platano", "amarillo")
kiwi = Fruta("kiwi", "verde")

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



OBJETOS Y CLASES

EJEMPLO:

```
manzana = Fruta("manzana", "rojo")
platano = Fruta("platano", "amarillo")
kiwi = Fruta("kiwi", "verde")
```

```
c: print(platano.nombre, platano.color)
    print(kiwi.nombre, kiwi.color)
    print(manzana.nombre, manzana.color)
```

✓ 0.0s

```
platano amarillo
kiwi verde
manzana rojo
```



OBJETOS Y CLASES

EJEMPLO:

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

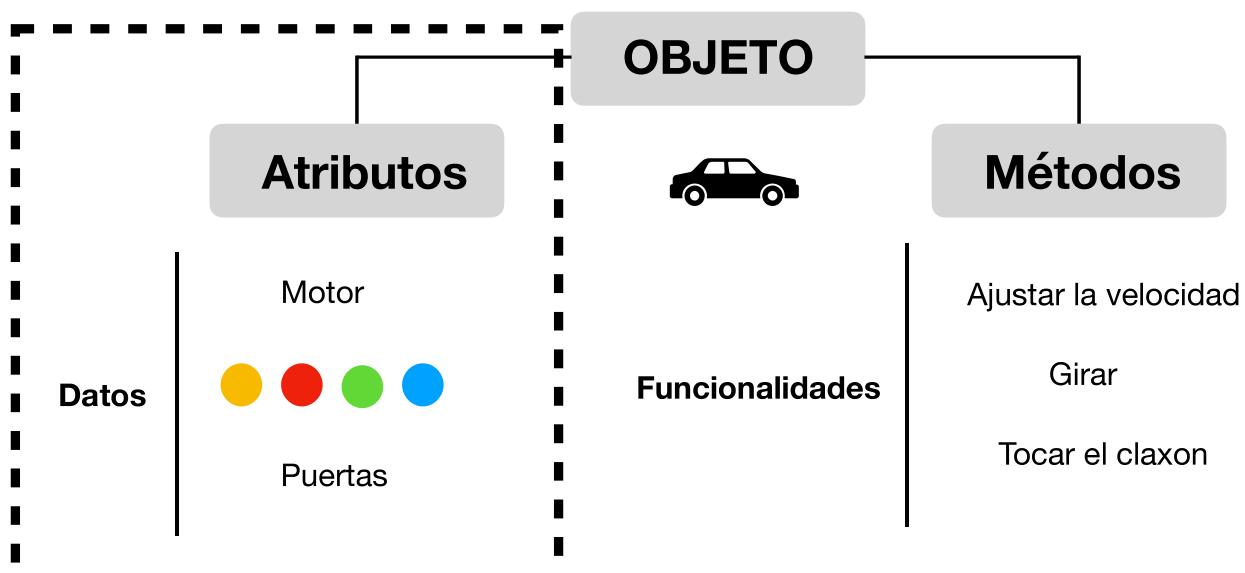
CLASE

```
manzana = Fruta("manzana", "rojo")
platano = Fruta("platano", "amarillo")
kiwi = Fruta("kiwi", "verde")
```

INSTANCIAS
/ OBJETOS

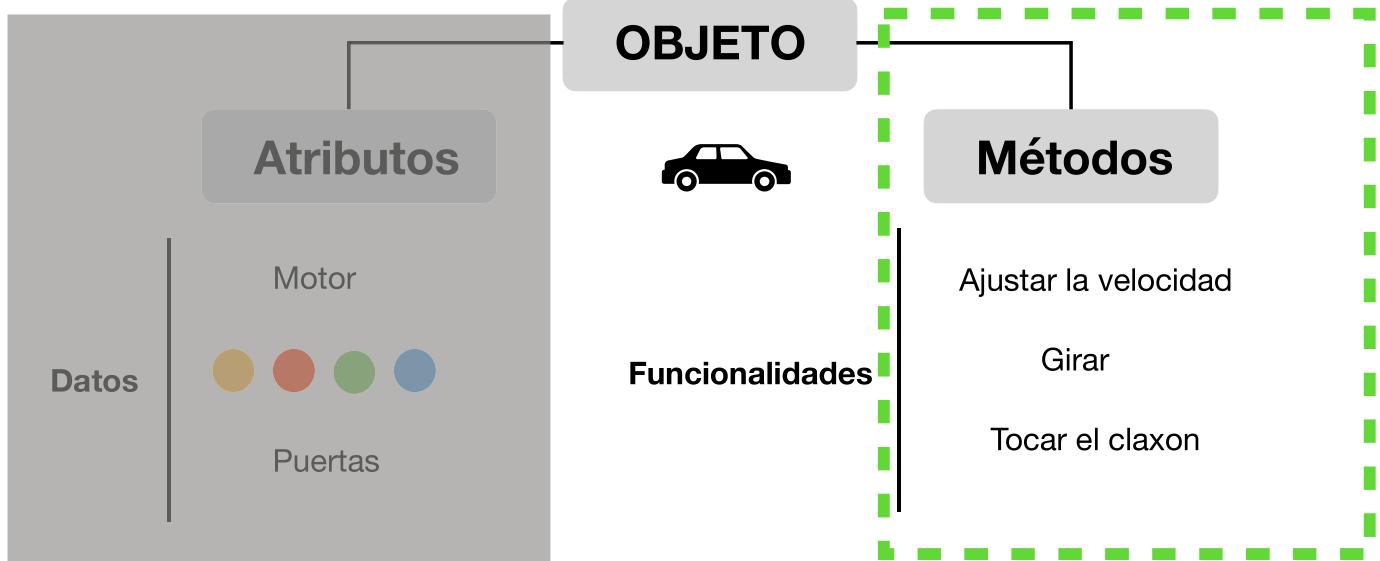


OBJETOS Y CLASES





OBJETOS Y CLASES



OBJETOS Y CLASES

```

class Fruta:
    def __init__(self, nombre, color):
        self.nombre = nombre
        self.color = color
    def details(self):
        print("mi " + self.nombre +
              " es " + self.color)

manzana = Fruta("manzana", "roja")
manzana.details()

✓ 0.0s
mi manzana es roja

```

Los métodos son funciones relacionadas con objetos



OBJETOS Y CLASES

```
class Fruta:  
    def __init__(self, nombre, color):  
        self.nombre = nombre  
        self.color = color  
    def details(self):  
        print("mi " + self.nombre +  
              " es " + self.color )
```

MÉTODO

Los métodos son
funciones relacionadas
con objetos



OBJETOS Y CLASES

```
class Fruta:  
    def __init__(self, nombre, color):  
        self.nombre = nombre  
        self.color = color  
    def details(self): MÉTODO  
        print("mi " + self.nombre +  
              " es " + self.color )
```

Los métodos son
funciones relacionadas
con objetos



OBJETOS Y CLASES

```
class Fruta:  
    def __init__(self, nombre, color):  
        self.nombre = nombre  
        self.color = color  
    def details(self):  
        print("mi " + self.nombre +  
              " es " + self.color )
```

MÉTODOS

Los métodos son funciones relacionadas con objetos



OBJETOS Y CLASES

```
class Fruta:  
    def __init__(self, nombre, color):  
        self.nombre = nombre  
        self.color = color  
    def details(self):  
        print("mi " + self.nombre +  
              " es " + self.color )
```

AL PASAR **SELF** A TODOS LOS MÉTODOS PODEMOS USAR LOS MISMOS ATRIBUTOS EN TODOS LOS MÉTODOS DE NUESTRA CLASE



OBJETOS Y CLASES

```
class Fruta:  
    def __init__(self, nombre, color):  
        self.nombre = nombre  
        self.color = color  
  
    def details(self):  
        print("mi " + self.nombre +  
              " es " + self.color )
```

A

B

AUNQUE AMBOS METODOS ESTAN HACIENDO COSAS COMPLETAMENTE DIFERENTES, PUEDEN ACCEDER A LOS MISMO ATRIBUTOS GRACIAS A SELF



OBJETOS Y CLASES

```
class Fruta:  
    def __init__(self, nombre, color):  
        self.nombre = nombre  
        self.color = color  
    def details(self):  
        print("mi " + self.nombre +  
              " es " + self.color )
```

```
manzana = Fruta("manzana", "roja")  
manzana.details()
```

Llamada al método

```
✓ 0.0s
```

```
mi manzana es roja
```



OBJETOS Y CLASES

ERRORES COMUNES — No usar self para un atributo —

```
class Fruta:
    def __init__(self, nombre, color):
        self.nombre = nombre
        self.color = color
        caducidad = "01.03.2023"
    def details(self):
        print("mi " + self.nombre +
              " es " + self.color )
        print("caduca el ", caducidad)

manzana = Fruta("manzana", "roja")
manzana.details()
```



OBJETOS Y CLASES

ERRORES COMUNES — No usar self para un atributo —

```
class Fruta:
    def __init__(self, nombr
        self.nombre = nombr
        self.color = color
        caducidad = "01.03.
    def details(self):
        print("mi " + self.
              " es " + self
        print("caduca el ",

manzana = Fruta("manzana", "roja")
manzana.details()
```

mi manzana es roja

```
NameError                                 Traceback (most recent call last)
Cell In[12], line 12
      9         print("caduca el ", caducidad)
     11 manzana = Fruta("manzana", "roja")
--> 12 manzana.details()

Cell In[12], line 9, in Fruta.details(self)
      6     def details(self):
      7         print("mi " + self.nombre +
      8               " es " + self.color )
--> 9         print("caduca el ", caducidad)

NameError: name 'caducidad' is not defined
```



OBJETOS Y CLASES

ERRORES COMUNES

— No usar self para un atributo —

```

class Fruta:
    def __init__(self, nombre, color):
        self.nombre = nombre
        self.color = color
        self.caducidad = "01.03.2023"
    def details(self):
        print("mi " + self.nombre +
              " es " + self.color)
        print("caduca el ", self.caducidad)

manzana = Fruta("manzana", "roja")
manzana.details()

```

mi manzana es roja

Traceback (most recent call last):

NameError: name 'caducidad' is not defined

Caducidad es una variable local de `__init__`. Al no ser global el metodo details no sabe de su existencia



OBJETOS Y CLASES

— SOLUCIÓN —

```

class Fruta:
    def __init__(self, nombre, color):
        self.nombre = nombre
        self.color = color
        self.caducidad = "01.03.2023"
    def details(self):
        print("mi " + self.nombre +
              " es " + self.color)
        print("caduca el ", self.caducidad)

manzana = Fruta("manzana", "roja")
manzana.details()

```

✓ 0.0s

mi manzana es roja
caduca el 01.03.2023

Convertir `caducidad` en un atributo

```

class Fruta:
    def __init__(self, nombre, color):
        self.nombre = nombre
        self.color = color
    def details(self):
        self.caducidad = "01.03.2023"
        print("mi " + self.nombre +
              " es " + self.color)
        print("caduca el ", self.caducidad)

manzana = Fruta("manzana", "roja")
manzana.details()

```

✓ 0.0s

mi manzana es roja
caduca el 01.03.2023

Definir `caducidad` dentro del metodo adecuado



MÉTODO __INIT__

```
class mi_clase:  
    def __init__(self):  
        self.attrib = []
```

- Es ejecutado automáticamente con cada nueva instancia de una clase (No necesitamos llamarlo como al resto de métodos)
- Es donde inicializamos los atributos
- Tiene el nombre reservado



MÉTODO __INIT__

```
class Fruta:  
    def __init__(self, clr):  
        self.color = clr  
  
manzana = Fruta("roja")
```

```
class mi_clase:  
    def __init__(self):  
        self.attrib = []
```

self = el objeto *manzana*

self = the *apple* object itself



MÉTODO __INIT__

```
class Fruta:
    def __init__(self, clr):
        self.color = clr

platano = Fruta("amarillo")
```

```
class mi_clase:
    def __init__(self):
        self.attrib = []
```

self = el objeto *platano*

self = the *banana* object itself



EN OTROS LENGUAJES DE PROGRAMACIÓN...

Javascript

```
function Fruta(nombre, clr) {
    this.nombre = nombre;
    this.color = clr;
}
```

Python

```
class Fruta:
    def __init__(self, nombre, clr):
        self.nombre = nombre
        self.color = clr
```

- **this**

- **función constructora**

- **self**

- **class**



ESTO ES LO QUE CONFORMA LA PROGRAMACION ORIENTADA A OBJETOS

- SET DE PRINCIPIOS DE PROGRAMACIÓN
- TRABAJAMOS CON OBJETOS EN VEZ DE CON DATOS Y PROCEDIMIENTOS SEPARADOS
- UNIMOS DATA Y FUNCIONALIDAD EN UNA SOLA ESTRUCTURA CREANDO OBJETOS QUE INTERACTUAN ENTRE ELLOS



ESTO ES LO QUE CONFORMA LA PROGRAMACION ORIENTADA A OBJETOS



CONQUER
BLOCKS

**CONQUER
BLOCKS**

PYTHON

PROGRAMACION ORIENTEADA
A OBJETOS (POO)

CONQUERBLOCKS



OBJETOS Y CLASES

CLASES →

Representaciones del mundo
real

Ejemplo:

COCHE





OBJETOS Y CLASES

OBJETOS →

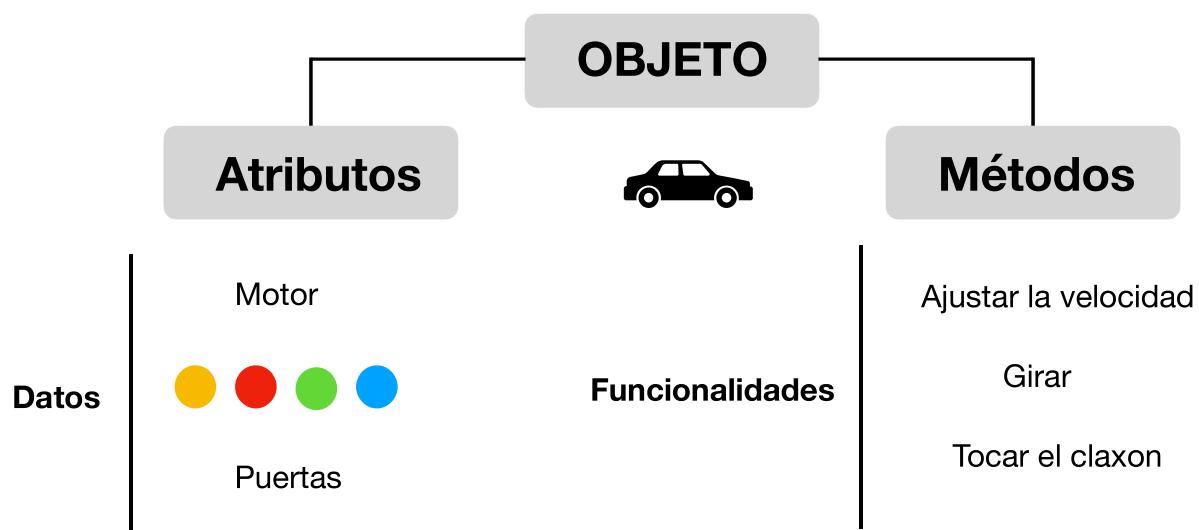
Una instancia de esa clase

Ejemplo:

MI
COCHE



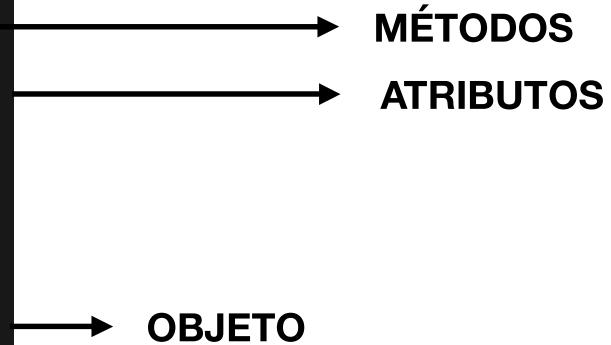
OBJETOS Y CLASES





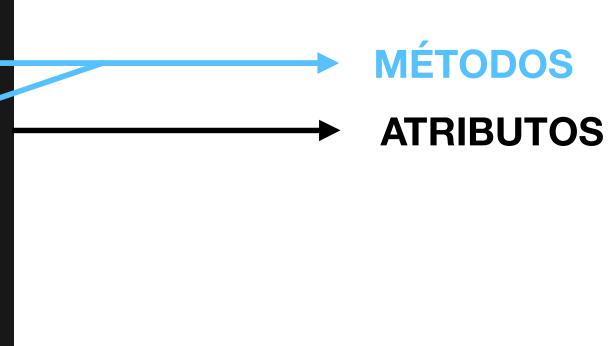
PROGRAMACION ORIENTADA A OBJETOS (POO)

```
class Fruta:  
    def __init__(self, nombre, color):  
        self.nombre = nombre  
        self.color = color  
        self.caducidad = "01.03.2023"  
    def details(self):  
        print("mi " + self.nombre +  
             " es " + self.color )  
        print("caduca el ", self.caducidad)  
  
manzana = Fruta("manzana", "roja")  
manzana.details()  
  
✓ 0.0s  
  
mi manzana es roja  
caduca el 01.03.2023
```



PROGRAMACION ORIENTADA A OBJETOS (POO)

```
class Fruta:  
    def __init__(self, nombre, color):  
        self.nombre = nombre  
        self.color = color  
        self.caducidad = "01.03.2023"  
    def details(self):  
        print("mi " + self.nombre +  
             " es " + self.color )  
        print("caduca el ", self.caducidad)  
  
manzana = Fruta("manzana", "roja")  
manzana.details()  
  
✓ 0.0s  
  
mi manzana es roja  
caduca el 01.03.2023
```





PROGRAMACION ORIENTADA A OBJETOS (POO)

```
class Fruta:  
    def __init__(self, nombre, color):  
        self.nombre = nombre  
        self.color = color  
        self.caducidad = "01.03.2023"  
    def details(self):  
        print("mi " + self.nombre +  
              " es " + self.color)  
        print("caduca el ", self.caducidad)  
  
manzana = Fruta("manzana", "roja")  
manzana.details()  
  
✓ 0.0s  
  
mi manzana es roja  
caduca el 01.03.2023
```

Diagrama de los componentes de la clase Fruta:

- MÉTODOS: Apuntan a los métodos `__init__` y `details`.
- ATRIBUTOS: Apuntan a las variables `nombre`, `color` y `caducidad`.
- OBJETO: Apunta al objeto `manzana` creado con `Fruta("manzana", "roja")`.



PROGRAMACION ORIENTADA A OBJETOS (POO)

```
class Fruta:  
    def __init__(self, nombre, color):  
        self.nombre = nombre  
        self.color = color  
        self.caducidad = "01.03.2023"  
    def details(self):  
        print("mi " + self.nombre +  
              " es " + self.color)  
        print("caduca el ", self.caducidad)  
  
manzana = Fruta("manzana", "roja")  
manzana.details()  
  
✓ 0.0s  
  
mi manzana es roja  
caduca el 01.03.2023
```

Diagrama de los componentes de la clase Fruta:

- MÉTODOS: Apuntan a los métodos `__init__` y `details`.
- ATRIBUTOS: Apuntan a las variables `nombre`, `color` y `caducidad`.
- OBJETO: Apuntan al objeto `manzana` creado con `Fruta("manzana", "roja")`.



CUATRO PILARES DEL POO

1. ABSTRACCIÓN

Simplificación y representación de entidades del mundo real en forma de objetos en el código. Estos encapsulan datos y comportamientos, permitiendo al programador enfocarse en lo que el objeto hace sin preocuparse del cómo.

2. ENCAPSULAMIENTO:

Agrupamiento de datos (atributos) y métodos (funciones) relacionados en un mismo objeto. La idea es que el objeto actúe como una cápsula que oculta los detalles internos y revela solo una interfaz para interactuar con él. De esta manera, se mejora la seguridad y se evita que el código externo modifique directamente el estado interno del objeto.

3. HERENCIA:

La herencia permite que una clase (subclase) herede los atributos y métodos de otra clase (superclase). La subclase puede extender o modificar la funcionalidad de la superclase y agregar nuevos atributos o métodos. Esto facilita la reutilización de código y la jerarquización.

4. POLIMORFISMO:

El polimorfismo es la capacidad de diferentes objetos de una jerarquía de clases para responder a una misma llamada de método de manera diferente. El polimorfismo permite que objetos de diferentes clases puedan ser tratados de manera uniforme, pero actúen de forma diferente según su propia implementación. Esto aumenta la flexibilidad del código y facilita la creación de interfaces genéricas que trabajen con múltiples clases.



```
class Coche:
    def __init__(self, marca, modelo, anio):
        """Inicializamos atributos del coche"""
        self.marca = marca
        self.modelo = modelo
        self.anio = anio

    def crear_descripcion(self):
        """Devolver una descripcion bien formateada"""
        nombre_extenso = str(self.anio) + " " + self.marca + " " + self.modelo
        return nombre_extenso

mi_coche = Coche("audi", "a4", "2016")
print(mi_coche.crear_descripcion())

```

✓ 0.0s
2016 audi a4



Añadimos un atributo con un valor por defecto:

```
class Coche:
    def __init__(self, marca, modelo, anio):
        """Inicializamos atributos del coche"""
        self.marca = marca
        self.modelo = modelo
        self.anio = anio
        self.cuenta_kilometros = 0

    def crear_descripcion(self):
        """Devolver una descripcion bien formateada"""
        nombre_extenso = str(self.anio) + " " + self.marca + " " + self.modelo
        return nombre_extenso

mi_coche = Coche("audi", "a4", "2016")
print(mi_coche.crear_descripcion())
print(mi_coche.cuenta_kilometros)

✓ 0.0s
2016 audi a4
0
```



Añadimos un método para leer el cuentakilómetros

```
class Coche:
    def __init__(self, marca, modelo, anio):
        """Inicializamos atributos del coche"""
        self.marca = marca
        self.modelo = modelo
        self.anio = anio
        self.cuenta_kilometros = 0

    def crear_descripcion(self):
        pass
        #--- snip --

    def leer_cuentakilometros(self):
        """Imprime el valor de los kilometros recorridos"""
        print("Este coche ha recorrido un total de", self.cuenta_kilometros, "km")

mi_coche = Coche("audi", "a4", "2016")
mi_coche.leer_cuentakilometros()

✓ 0.0s
Este coche ha recorrido un total de 0 km
```



Podemos modificar el atributo de kilometraje directamente...

```

class Coche:
    def __init__(self, marca, modelo, anio):
        """Inicializamos atributos del coche"""
        self.marca = marca
        self.modelo = modelo
        self.anio = anio
        self.cuenta_kilometros = 0

    def crear_descripcion(self):
        pass
        #-- snip --

    def leer_cuentakilometros(self):
        """Imprime el valor de los kilmetros recorridos"""
        print("Este coche ha recorrido un total de", self.cuenta_kilometros, "km")

    mi_coche = Coche("audi", "a4", "2016")
    mi_coche.cuenta_kilometros = 100
    mi_coche.leer_cuentakilometros()

✓ 0.0s
Este coche ha recorrido un total de 100 km
  
```



Podemos modificar el atributo de kilometraje mediante un método...

```

class Coche:
    def __init__(self, marca, modelo, anio):
        """Inicializamos atributos del coche"""
        self.marca = marca
        self.modelo = modelo
        self.anio = anio
        self.cuenta_kilometros = 0

    def crear_descripcion(self):
        pass
        #-- snip --

    def leer_cuentakilometros(self):
        """Imprime el valor de los kilmetros recorridos"""
        print("Este coche ha recorrido un total de", self.cuenta_kilometros, "km")

    def actualiza_cuentakilometros(self, kilometros):
        """Actualiza el valor del cuentakilometros al valor dado"""
        self.cuenta_kilometros = kilometros

    mi_coche = Coche("audi", "a4", "2016")
    mi_coche.actualiza_cuentakilometros(100)
    mi_coche.leer_cuentakilometros()

✓ 0.0s
Este coche ha recorrido un total de 100 km
  
```



Esto es mucho más conveniente ya que así podemos asegurarnos de que no se cambie el kilometraje a uno menor del actual

```
class Coche:
    def __init__(self, marca, modelo, anio):
        """Inicializamos atributos del coche"""
        self.marca = marca
        self.modelo = modelo
        self.anio = anio
        self.cuenta_kilometros = 0

    def crear_descripcion(self):
        pass
        #-- snip --

    def leer_cuentakilometros(self):
        """Imprime el valor de los kilometros recorridos"""
        print("Este coche ha recorrido un total de", self.cuenta_kilometros, "km")
```

```
def actualiza_cuentakilometros(self, kilometros):
    """Actualiza el valor del cuentakilometros al valor dado"""
    if kilometros >= self.cuenta_kilometros:
        self.cuenta_kilometros = kilometros
    else:
        print("No puedes disminuir el kilometraje")
```

```
mi_coche = Coche("audi", "a4", "2016")
mi_coche.actualiza_cuentakilometros(100)
mi_coche.leer_cuentakilometros()
mi_coche.actualiza_cuentakilometros(50)
mi_coche.leer_cuentakilometros()

✓ 0.0s
```

Este coche ha recorrido un total de 100 km
 No puedes disminuir el kilometraje
 Este coche ha recorrido un total de 100 km



Incrementar el valor de un atributo mediante un método...

```
class Coche:
    def __init__(self, marca, modelo, anio):
        """Inicializamos atributos del coche"""
        self.marca = marca
        self.modelo = modelo
        self.anio = anio
        self.cuenta_kilometros = 0

    def crear_descripcion(self):
        pass
        #-- snip --

    def leer_cuentakilometros(self):
        """Imprime el valor de los kilometros recorridos"""
        print("Este coche ha recorrido un total de", self.cuenta_kilometros, "km")

    def actualiza_cuentakilometros(self, kilometros):
        """Actualiza el valor del cuentakilometros al valor dado"""
        if kilometros >= self.cuenta_kilometros:
            self.cuenta_kilometros = kilometros
        else:
            print("No puedes disminuir el kilometraje")
```

```
def aumentar_cuentakilometros(self, kilometros):
    """Suma el valor dado al cuentakilometros """
    self.cuenta_kilometros += kilometros
```

```
mi_coche_usado = Coche("audi", "a4", "2016")
mi_coche_usado.actualiza_cuentakilometros(23500)
mi_coche_usado.leer_cuentakilometros()
mi_coche_usado.aumentar_cuentakilometros(100)
mi_coche_usado.leer_cuentakilometros()

✓ 0.0s
```

Este coche ha recorrido un total de 23500 km
 Este coche ha recorrido un total de 23600 km



Y si ahora tengo un coche eléctrico?

Puedo crear otra clase CocheElectrico y añadirle los mismo atributos que a la clase Coche

```
class Coche:  
    def __init__(self, marca, modelo, anio):  
        """Inicializamos atributos del coche"""  
        self.marca = marca  
        self.modelo = modelo  
        self.anio = anio  
        self.cuenta_kilometros = 0  
  
    def crear_descripcion(self):  
        pass  
        #-- snip --  
  
    def leer_cuentakilometros(self):  
        """Imprime el valor de los kilometros recorridos"""  
        print("Este coche ha recorrido un total de", self.cuenta_kilometros, "km")  
  
    def actualiza_cuentakilometros(self, kilometros):  
        """Actualiza el valor del cuentakilometros al valor dado"""  
        if kilometros >= self.cuenta_kilometros:  
            self.cuenta_kilometros = kilometros  
        else:  
            print("No puedes disminuir el kilometraje")
```



Y si ahora tengo un coche eléctrico?

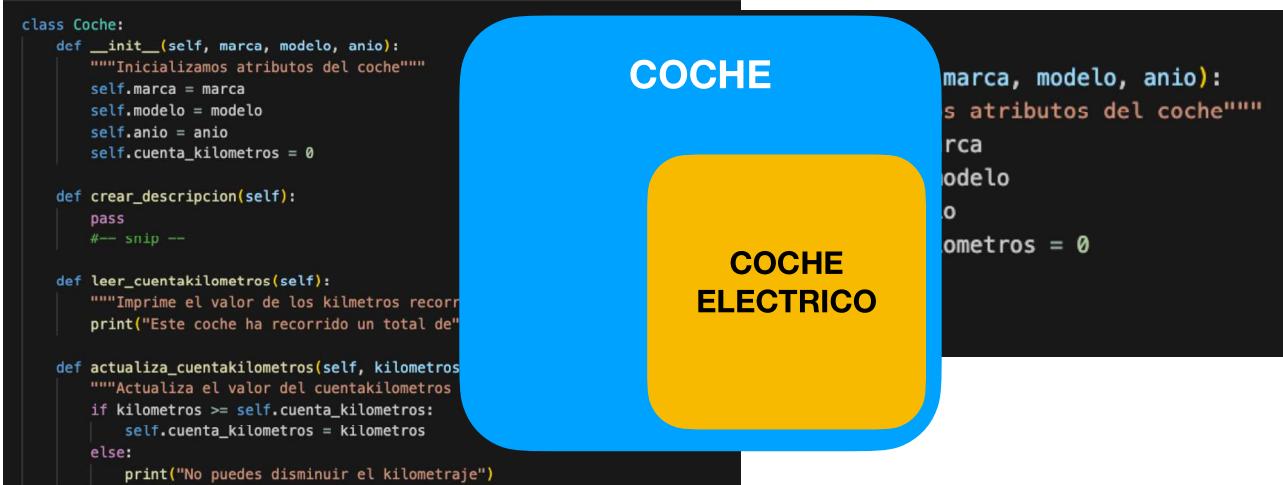
Puedo crear otra clase CocheElectrico y añadirle los mismo atributos que a la clase Coche

```
class Coche:  
    def __init__(self, marca, modelo, anio):  
        """Inicializamos atributos del coche"""  
        self.marca = marca  
        self.modelo = modelo  
        self.anio = anio  
        self.cuenta_kilometros = 0  
  
    def crear_descripcion(self):  
        pass  
        #-- snip --  
  
    def leer_cuentakilometros(self):  
        """Imprime el valor de los kilometros recorridos"""  
        print("Este coche ha recorrido un total de", self.cuenta_kilometros)  
  
    def actualiza_cuentakilometros(self, kilometros):  
        """Actualiza el valor del cuentakilometros al valor dado"""  
        if kilometros >= self.cuenta_kilometros:  
            self.cuenta_kilometros = kilometros  
        else:  
            print("No puedes disminuir el kilometraje")  
  
class CocheElectrico:  
    def __init__(self, marca, modelo, anio):  
        """Inicializamos atributos del coche"""  
        self.marca = marca  
        self.modelo = modelo  
        self.anio = anio  
        self.cuenta_kilometros = 0  
  
    ...
```



Y si ahora tengo un coche eléctrico?

Puedo crear otra clase CocheElectrico y añadirle los mismo atributos que a la clase Coche --->>> esto es muy INEFICIENTE



Y si ahora tengo un coche eléctrico? Usaremos la *herencia* para crear una clase *hija*

PARENT CLASS (SUPERCLASS) / CLASE PADRE

```

class Coche:
    def __init__(self, marca, modelo, anio):
        """Inicializamos atributos del coche"""
        self.marca = marca
        self.modelo = modelo
        self.anio = anio
        self.cuenta_kilometros = 0

    def crear_descripcion(self):
        """Devolver una descripción bien formateada"""
        nombre_extenso = str(self.anio) + " " + self.marca + " " + self.modelo
        return nombre_extenso

    def leer_cuentakilometros(self):
        """Imprime el valor de los kilometros recorridos"""
        print("Este coche ha recorrido un total de", self.cuenta_kilometros, "km")

    def actualiza_cuentakilometros(self, kilometros):
        """Actualiza el valor del cuentakilometros al valor dado"""
        if kilometros >= self.cuenta_kilometros:
            self.cuenta_kilometros = kilometros
        else:
            print("No puedes disminuir el kilometraje")

    def aumentar_cuentakilometros(self, kilometros):
        """Suma el valor dado al cuentakilometros """
        self.cuenta_kilometros += kilometros

```

CHILD CLASS (SUBCLASS) / CLASE HIJO

```

class CocheElectrico(Coche):
    """Representa aspectos de un coche,
    especifico para coches electricos"""
    def __init__(self, marca, modelo, anio):
        super().__init__(marca, modelo, anio)

mi_tesla = CocheElectrico("tesla", "modelo s", 2016)
print(mi_tesla.crear_descripcion())

```

✓ 0.0s
2016 tesla modelo s



Y si ahora tengo un coche eléctrico? Usaremos la *herencia* para crear una clase *hija*

CHILD CLASS (SUBCLASS) / CLASE HIJO

```
class CocheElectrico(Coche):
    """Representa aspectos de un coche,
    especifico para coches electricos"""
    def __init__(self, marca, modelo, anio):
        super().__init__(marca, modelo, anio)

    mi_tesla = CocheElectrico("tesla", "modelo s", 2016)
    print(mi_tesla.crear_descripcion())

✓ 0.0s

2016 tesla modelo s
```



Y si ahora tengo un coche eléctrico? Usaremos la *herencia* para crear una clase *hija*

CHILD CLASS (SUBCLASS) / CLASE HIJO

```
class CocheElectrico(Coche): Nombre de la superclase
    """Representa aspectos de un coche,
    especifico para coches electricos"""
    def __init__(self, marca, modelo, anio):
        super().__init__(marca, modelo, anio)
        Función que indica a python que conecte con la superclase
    mi_tesla = CocheElectrico("tesla", "modelo s", 2016)
    print(mi_tesla.crear_descripcion())

✓ 0.0s

2016 tesla modelo s
```



Y si ahora tengo un coche eléctrico? Usaremos la *herencia* para crear una clase *hija*

CHILD CLASS (SUBCLASS) / CLASE HIJO

```
class CocheElectrico(Coche):
    """Representa aspectos de un coche,
    especifico para coches electricos"""
    def __init__(self, marca, modelo, anio):
        super().__init__(marca, modelo, anio)
```

Indica a python que llame a la función `__init__` de la superclase.
Esto concede a la subclase todos los atributos de la superclase

HERENCIA



Y si ahora tengo un coche eléctrico? Usaremos la *herencia* para crear una clase *hija*

CHILD CLASS (SUBCLASS) / CLASE HIJO

```
class CocheElectrico(Coche):
    """Representa aspectos de un coche,
    especifico para coches electricos"""
    def __init__(self, marca, modelo, anio):
        super().__init__(marca, modelo, anio)

    mi_tesla = CocheElectrico("tesla", "modelo s", 2016)
    print(mi_tesla.crear_descripcion())
    ✓ 0.0s También ha heredado los métodos de la superclase

2016 tesla modelo s
```



Y si ahora tengo un coche eléctrico? Usaremos la *herencia* para crear una clase *hija*

CHILD CLASS (SUBCLASS) / CLASE HIJO

```

class CocheElectrico(Coche):
    """Representa aspectos de un coche,
    especifico para coches electricos"""
    def __init__(self, marca, modelo, anio):
        """ Inicializar atributos de superclase.
        Despues inicializar atributos especificos
        del coche electrico """
        super().__init__(marca, modelo, anio)
        self.tamanio_bateria = 70

    def crear_descripcion(self):
        """Imprime una descripcion de la bateria"""
        print("Este coche tiene una bateria de tamaño",
              self.tamanio_bateria, "kWh")

mi_tesla = CocheElectrico("tesla", "modelo s", 2016)
print(mi_tesla.crear_descripcion())
mi_tesla.descripcion_bateria()

✓ 0.0s

2016 tesla modelo s

```

NOTA: Para que todo funcione correctamente, la subclase deberá estar después de la superclase dentro del archivo



Una vez nuestro coche eléctrico ha heredado los atributos y métodos de la superclase coche, podemos añadir métodos que solo apliquen a los coches eléctricos

```

class CocheElectrico(Coche):
    """Representa aspectos de un coche,
    especifico para coches electricos"""
    def __init__(self, marca, modelo, anio):
        """ Inicializar atributos de superclase.
        Despues inicializar atributos especificos
        del coche electrico """
        super().__init__(marca, modelo, anio)
        self.tamanio_bateria = 70

    def crear_descripcion(self):
        """Imprime una descripcion de la bateria"""
        print("Este coche tiene una bateria de tamaño",
              self.tamanio_bateria, "kWh")

    def descripcion_bateria(self):
        """Imprime una descripcion de la bateria"""
        print("Este coche tiene una bateria de tamaño",
              self.tamanio_bateria, "kWh")

mi_tesla = CocheElectrico("tesla", "modelo s", 2016)
print(mi_tesla.crear_descripcion())
mi_tesla.descripcion_bateria()

✓ 0.0s

2016 tesla modelo s
Este coche tiene una bateria de tamaño 70 kWh

```



También podemos sobreescribir métodos de la superclase

```
class Coche:
    def __init__(self, marca, modelo, anio):
        """Inicializamos atributos del coche"""
        self.marca = marca
        self.modelo = modelo
        self.anio = anio
        self.deposito = 0

    def llenar_deposito(self):
        """Simula el llenado del deposito del coche"""
        self.deposito = 100
        print("El deposito esta a", self.deposito)
```

```
class CocheElectrico(Coche):
    """Representa aspectos de un coche,
    especifico para coches electricos"""

    def __init__(self, marca, modelo, anio):
        """ Inicializar atributos de superclase.
        Despues inicializar atributos especificos
        del coche electrico """
        super().__init__(marca, modelo, anio)
        self.tamano_bateria = 70

    def llenar_deposito(self):
        """Los coches electricos no tienes deposito"""
        print("Este coche no tiene un deposito")
```

```
mi_audi = Coche("audi", "a4", 2015)
mi_audi.llenar_deposito()
mi_tesla = CocheElectrico("tesla", "modelo s", 2016)
mi_tesla.llenar_deposito()

✓ 0.0s

El deposito esta a 100
Este coche no tiene un deposito
```



También podemos sobreescribir métodos de la superclase

```
class Coche:
    def __init__(self, marca, modelo, anio):
        """Inicializamos atributos del coche"""
        self.marca = marca
        self.modelo = modelo
        self.anio = anio
        self.deposito = 0

    def llenar_deposito(self):
        """Simula el llenado del deposito"""
        self.deposito = 100
        print("El deposito esta a", self.deposito)
```

POLIMORFISMO

```
class CocheElectrico(Coche):
    """Representa aspectos de un coche,
    especifico para coches electricos"""

    def __init__(self, marca, modelo, anio):
        """ Inicializar atributos de superclase.
        Despues inicializar atributos especificos
        del coche electrico """
        super().__init__(marca, modelo, anio)
        self.tamano_bateria = 70

    def llenar_deposito(self):
        """Los coches electricos no tienes deposito"""
        print("Este coche no tiene un deposito")
```

```
mi_audi = Coche("audi", "a4", 2015)
mi_audi.llenar_deposito()
mi_tesla = CocheElectrico("tesla", "modelo s", 2016)
mi_tesla.llenar_deposito()

✓ 0.0s

El deposito esta a 100
Este coche no tiene un deposito
```



Cuando estemos modelizando un elemento del mundo real puede que nos encontremos añadiendo cada vez más detalles dentro de una clase. Quizás nos encontremos con una lista creciente de atributos y métodos que hacen que nuestros archivos sean muy extensos.

Ejemplo: Si continuamos añadiendo detalles a la clase CocheElectrico puede que notemos que estamos añadiendo muchos atributos y métodos relacionados con la batería del coche.

En estos momentos toca parar y mover estos atributos y métodos a una clase nueva llamada batería...

```
class Coche:
    def __init__(self, marca, modelo, anio):
        """Inicializamos atributos del coche"""
        self.marca = marca
        self.modelo = modelo
        self.anio = anio
        self.deposito = 0

    def llenar_deposito(self):
        """Simula el llenado del deposito del coche"""
        self.deposito = 100
        print("El deposito esta a", self.deposito)

class Bateria:
    """Un intento simple de modelizar
    una bateria"""

    def __init__(self, tamanio_bateria = 70):
        """Inicializamos los atributos de la bateria"""
        self.tamanio_bateria = tamanio_bateria

    def describir_bateria(self):
        """Imprimir el tamaño de la bateria"""
        print("Este coche tiene una bateria de tamaño",
              self.tamanio_bateria,"-kWh.")
```

```
class CocheElectrico(Coche):
    """Representa aspectos de un coche,
    especifico para coches electricos"""
    def __init__(self, marca, modelo, anio):
        """ Inicializar atributos de superclase.
        Despues inicializar atributos especificos
        del coche electrico """
        super().__init__(marca, modelo, anio)
        self.bateria = Bateria() Instancia como
                            atributo

mi_tesla = CocheElectrico("tesla", "modelo s", 2016)
mi_tesla.llenar_deposito()
mi_tesla.bateria.describir_bateria()

✓ 0.0s

El deposito esta a 100
Este coche tiene una bateria de tamaño 70 -kWh.
```

```

class Coche:
    def __init__(self, marca, modelo, anio):
        """Inicializamos atributos del coche"""
        self.marca = marca
        self.modelo = modelo
        self.anio = anio
        self.deposito = 0

    def llenar_deposito(self):
        """Simula llenar el deposito"""
        self.deposito = 100
        print("El deposito esta a 100")

class Bateria:
    """Un intento simple de modelizar
    una bateria"""

    def __init__(self, tamanio_bateria = 70):
        """Inicializamos los atributos de la bateria"""
        self.tamanio_bateria = tamanio_bateria

    def describir_bateria(self):
        """Imprimir el tamaño de la bateria"""
        print("Este coche tiene una bateria de tamaño",
              self.tamanio_bateria,"-kWh.")

```

```

class CocheElectrico(Coche):
    """Representa aspectos de un coche,
    especifico para coches electricos"""
    def __init__(self, marca, modelo, anio):
        """ Inicializar atributos de superclase.
        Despues inicializar atributos especificos
        del coche electrico """
        super().__init__(marca, modelo, anio)
        self.bateria = Bateria()

mi_tesla = CocheElectrico("tesla", "modelo s", 2016)

```

TAMBIEN PODRIAMOS AÑADIR UN METODO QUE CALCULE EL RANGO DE RECORRIDO QUE LE QUEDA AL COCHE DAD LA CARGA DE LA BATERIA...

El deposito esta a 100
Este coche tiene una bateria de tamaño 70 -kWh.



TAMBIEN PODRIAMOS AÑADIR UN METODO QUE CALCULE EL RANGO DE RECORRIDO QUE LE QUEDA AL COCHE DADA LA CARGA DE LA BATERIA...

¿Ese método correspondería a la clase Batería o al CocheEléctrico?

Si tenemos solo un coche quizás podríamos poner el método en la clase Bateria.

Pero si tenemos una linea de coches diferentes, cada uno puede consumir la batería a distinto ritmo. Podría ser conveniente que el método lea el estado de la batería y calcule el rango dado el modelo —> debería estar en la clase CocheElectrico



TAMBIEN PODRIAMOS AÑADIR UN METODO QUE CALCULE EL RANGO DE RECORRIDO QUE LE QUEDA AL COCHE DADA LA CARGA DE LA BATERIA...

Estas cuestiones nos llevan a un punto interesante en el crecimiento como programadores —>>> Estamos en un nivel de lógica superior.

No estamos pensando a un nivel sintáctico. No pensamos en Python si no en como modelizar el mundo a nuestro alrededor con código.



ASÍ QUE...

Algunos enfoques son más eficientes que otros, pero se necesita práctica para encontrar las representaciones más eficientes. Si tu código está funcionando como deseas, ¡lo estás haciendo bien!

No te desanimes si descubres que estás desglosando tus clases y reescribiéndolas varias veces utilizando enfoques diferentes. En la búsqueda de escribir un código preciso y eficiente, todos pasan por este proceso.

CONQUER
BLOCKS

**CONQUER
BLOCKS**

PYTHON

AMA

CONQUERBLOCKS



Hola **@lahelen** estaría bien un AMA de comprensión de listas para entenderlas mejor e ir poniendolas en práctica

Hola buenaas **@lahelen** para la proxima ama de python avanzado podrias explicar un poco mas lo de la recursividad de las funciones que aun no me a quedado claro de el todo.
Graciaas!

Hola **@lahelen** , de expresiones regulares vemos algo? Sino se puede ver en un ama ?

@lahelen podriamos ver en el proximo ama la construccion if `name == "main"`:
Es algo que no vimos pero la estuve utilizando a trabajar con diferentes al importarlos los modulos.



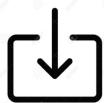
@lahelen podriamos ver en el proximo ama la construccion if `name == "main"`:
Es algo que no vimos pero la estuve utilizando a trabajar con diferentes al importarlos los modulos.



@lahelen podriamos ver en el proximo ama la construccion if `name == "main"`:
Es algo que no vimos pero la estuve utilizando a trabajar con diferentes al importarlos los modulos.

¿QUÉ ES `_name_`?

`_name_` es una **variable** especial que usamos al interactuar con **módulos**



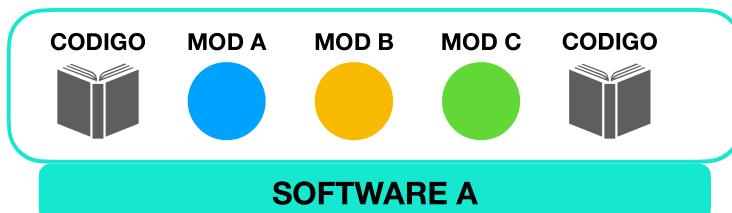
Nos ayuda a distinguir los módulos que han sido importados de los que no



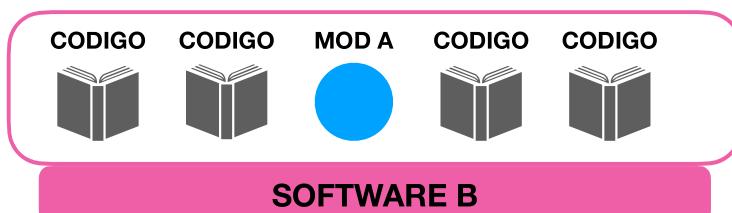


QUE ES UN MÓDULO...

Trozo de código **autocontenido, separable** del resto del código

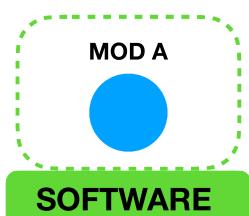


Un modulo de un software puede usarse en otro software

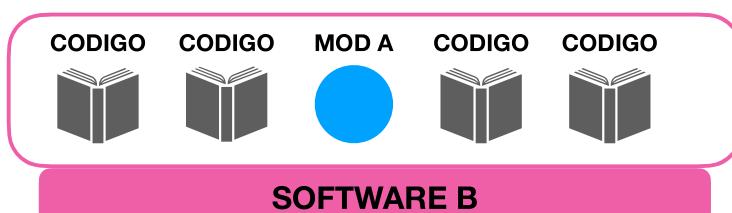


QUE ES UN MÓDULO...

Trozo de código **autocontenido, separable** del resto del código



Un modulo también puede ejecutarse de manera aislada





QUE ES UN MÓDULO...

Trozo de código **autocontenido, separable** del resto del código

Será un módulo todo aquello que...

1. se pueda re-utilizar en un software diferente
2. se pueda ejecutar de manera independiente
3. sea identificable entre distintas unidades

Por lo tanto todos los siguientes elementos son *módulos*



`__name__` nos ayuda a distinguir los módulos que han sido importados de los que no

A. librerías
funciones
clases
  
que importamos a nuestro código

B. el archivo o código de Python que estamos ejecutando en nuestra consola
 
donde estamos importando las cosas



mi_script.py × → Top Level Code

```
if __name__ == "__main__":
    print("esto es un modulo")
```

PROBLEMAS SALIDA CONSOLA DE DEPURACIÓN TERMINAL

```
(cblocks) MacBook-Pro-5:if_name_main Elena$ python mi_script.py
esto es un modulo
(cblocks) MacBook-Pro-5:if_name_main Elena$
```



mi_script.py × → Top Level Code

```
if __name__ == "__main__":
    import numpy
    arr1 = numpy.array([0,1,2])
    print("esto es un modulo")
```



```
mi_script.py × No es top level code  
if_name_main > mi_script.py > {} numpy  
1 import numpy  
2 arr1 = numpy.array([0,1,2])  
3 print("esto es un modulo")
```



```
mi_script.py × No es top level code  
if_name_main > mi_script.py > {} numpy  
1 import numpy  
2 arr1 = numpy.array([0,1,2])  
3 print("esto es un modulo")
```

Hemos tomado prestado un elemento que nunca hemos definido en nuestro código ya que pertenece a numpy



mi_script.py × No es top level code

```

1 if __name__ == "__main__":
2     print("estoy en el modulo")
3     print("esto es un modulo")

```

Esto es lo que __name__ nos va a ayudar a distinguir

Hemos tomado prestado un elemento que nunca hemos definido en nuestro código ya que pertenece a numpy

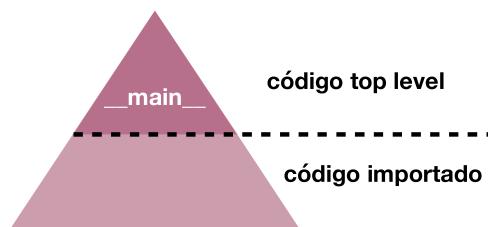


@lahelen podriamos ver en el proximo am la construccion `if name == 'main'`:
Es algo que no vimos pero la estuve utilizando a trabajar con diferentes al importarlos los modulos.

`__main__` representa el nombre (name) del **top level environment** donde el código está siendo ejecutado

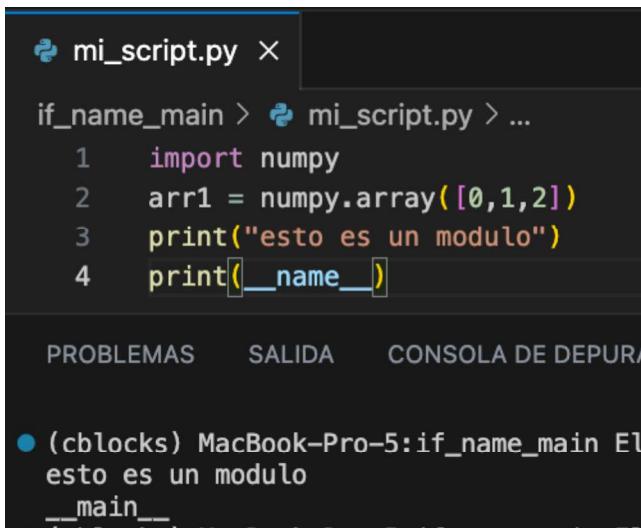
`if __name__ == "__main__"`

estamos comprobando si el código ejecutado es top level o no





`__main__` representa el nombre (name) del **top level environment** donde el código está siendo ejecutado

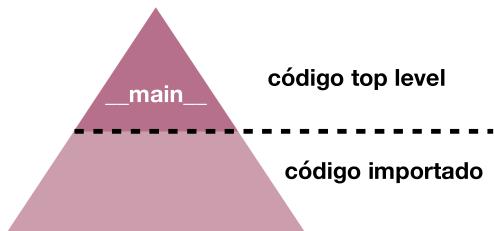


```
mi_script.py X

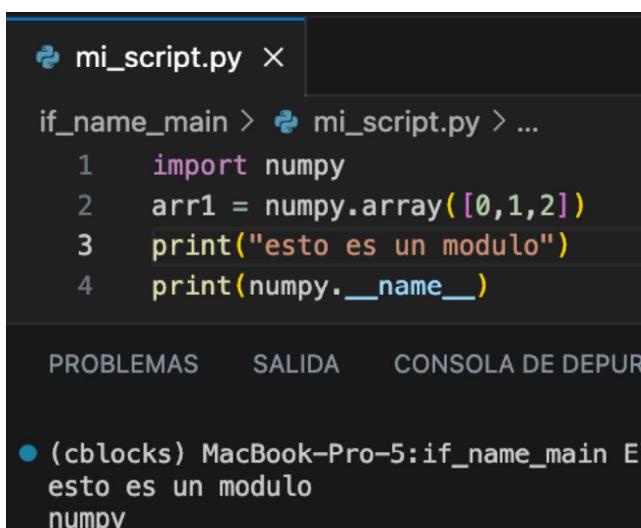
if_name_main > mi_script.py > ...
1 import numpy
2 arr1 = numpy.array([0,1,2])
3 print("esto es un modulo")
4 print(__name__)

PROBLEMAS SALIDA CONSOLA DE DEPURAR

● (cblocks) MacBook-Pro-5:if_name_main El
 esto es un modulo
 __main__
```



`__main__` representa el nombre (name) del **top level environment** donde el código está siendo ejecutado

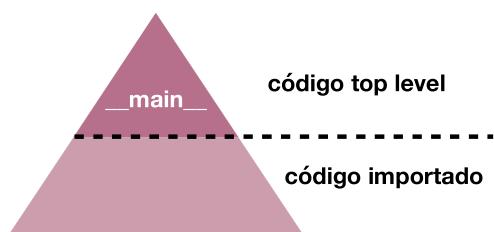


```
mi_script.py X

if_name_main > mi_script.py > ...
1 import numpy
2 arr1 = numpy.array([0,1,2])
3 print("esto es un modulo")
4 print(numpy.__name__)

PROBLEMAS SALIDA CONSOLA DE DEPURAR

● (cblocks) MacBook-Pro-5:if_name_main El
 esto es un modulo
 numpy
```





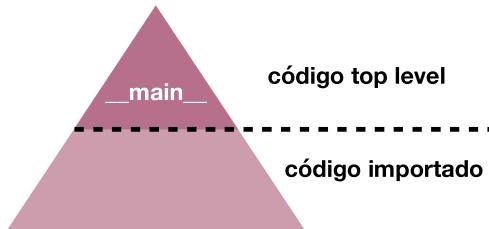
`__main__` representa el nombre (name) del **top level environment** donde el código está siendo ejecutado

```
mi_script.py ×

if __name__ == "__main__":
    import numpy as np
    arr1 = np.array([0,1,2])
    print("esto es un modulo")
    print(np.__name__)

PROBLEMAS SALIDA CONSOLA DE DEPURACION

● (cblocks) MacBook-Pro-5:if_name_main
  esto es un modulo
  numpy
○ (cblocks) MacBook-Pro-5:if_name_main
```



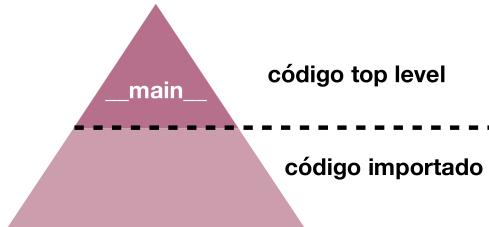
`__main__` representa el nombre (name) del **top level environment** donde el código está siendo ejecutado

```
mi_script.py ×

if __name__ == "__main__":
    # y con imports específicos...
    from datetime import timezone
    print(timezone.__name__)

PROBLEMAS SALIDA CONSOLA DE DEPURACION

● (cblocks) MacBook-Pro-5:if_name_main El nombre de la función o variable no se reconoce
● (cblocks) MacBook-Pro-5:if_name_main El nombre de la función o variable no se reconoce
○ (cblocks) MacBook-Pro-5:if_name_main El nombre de la función o variable no se reconoce
```



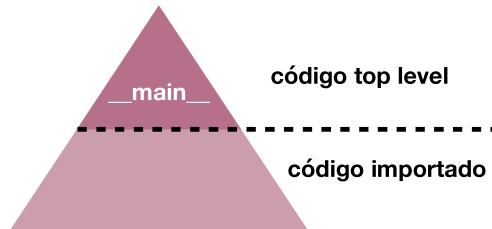


`__main__` representa el nombre (name) del **top level environment** donde el código está siendo ejecutado

```
mi_script.py X
if __name_main > mi_script.py > ...
7   # y con imports específicos...
8   from datetime import timezone
9   print(timezone.__name__)
10  print(datetime.__name__)

PROBLEMAS SALIDA CONSOLA DE DEPURACIÓN

(cblocks) MacBook-Pro-5:if_name_main Elena$ py
timezone
Traceback (most recent call last):
File "/Users/Elena/Desktop/Master Conquer Bl
ain/mi_script.py", line 10, in <module>
    print(datetime.__name__)
NameError: name 'datetime' is not defined
(cblocks) MacBook-Pro-5:if_name_main Elena$
```



`__main__` representa el nombre (name) del **top level environment** donde el código está siendo ejecutado

```
mi_script.py X
if __name_main > mi_script.py > ...
7   # y con imports específicos...
8   from datetime import timezone
9   import datetime
10  print(timezone.__name__)
11  print(datetime.__name__)
12

PROBLEMAS SALIDA CONSOLA DE DEPURACIÓN

● (cblocks) MacBook-Pro-5:if_name_main Elena
timezone
datetime
○ (cblocks) MacBook-Pro-5:if_name_main Elena
```

Regla general: `__name__` nos devolverá el nombre del archivo sin la extensión .py.

Usamos el nombre del archivo como nombre del modulo (ejemplo: `datetime`)



`__main__` representa el nombre (name) del *top level environment* donde el código está siendo ejecutado

```
mi_script.py X
if __name__ == "__main__":
    # y con imports específicos...
    from datetime import timezone
    import datetime
    print(timezone.__name__)
    print(datetime.__name__)

PROBLEMAS SALIDA CONSOLA DE DEPURACIÓN

● (cblocks) MacBook-Pro-5:if_name_main Elena
    timezone
    datetime
○ (cblocks) MacBook-Pro-5:if_name_main Elena
```

Si estamos importando una clase en específico, su nombre será el nombre de la clase (ejemplo: *timezone*)



¿PARA QUÉ USAMOS TODO ESTO?



```

para_importar.py ×
if_name_main > para_importar.py
1 def llamame():
2     print("¡Hola!")
3
4 llamame()

PROBLEMAS SALIDA CONSOLA

● (cblocks) MacBook-Pro-5:if_name_main Elena$ python3.9 para_ejecutar.py
¡Hola!
○ (cblocks) MacBook-Pro-5:if_name_main Elena$ █

para_ejecutar.py ×
if_name_main > para_ejecutar.py > llamame
1 from para_importar import llamame

PROBLEMAS SALIDA CONSOLA DE DEPURACIÓN TERMINAL

● (cblocks) MacBook-Pro-5:if_name_main Elena$ python3.9 para_ejecutar.py
¡Hola!
○ (cblocks) MacBook-Pro-5:if_name_main Elena$ █

```



```

para_importar.py ×
if_name_main > para_importar.py
1 def llamame():
2     print("¡Hola!")
3
4 llamame()

PROBLEMAS SALIDA CONSOLA

● (cblocks) MacBook-Pro-5:if_name_main Elena$ python3.9 para_ejecutar.py
¡Hola!
○ (cblocks) MacBook-Pro-5:if_name_main Elena$ █

para_ejecutar.py ×
if_name_main > para_ejecutar.py > llamame
1 from para_importar import llamame

PROBLEMAS SALIDA CONSOLA DE DEPURACIÓN TERMINAL

● (cblocks) MacBook-Pro-5:if_name_main Elena$ python3.9 para_ejecutar.py
¡Hola!
○ (cblocks) MacBook-Pro-5:if_name_main Elena$ █

```

Vemos el output de llamame() al ejecutar el script para_ejecutar.py, aunque en ningún momento hemos llamado a la función.



if __name__ == “__main__” nos sirve para evitar esto

The screenshot shows a code editor interface with two tabs: 'para_importar.py' and 'para_ejecutar.py'. The 'para_importar.py' tab contains the following code:

```
if __name__ == "__main__":
    def llamame():
        print("¡Hola!")
    if __name__ == "__main__":
        llamame()
```

The 'para_ejecutar.py' tab contains the following code:

```
from para_importar import llamame
llamame()
```

Below the tabs, there are navigation buttons: PROBLEMAS, SALIDA, CONSOLA DE DEPURACIÓN, and TERMINAL. The TERMINAL section shows the command-line output of running the script:

```
(cblocks) MacBook-Pro-5:if_name_main Elena$ python3.9 para_ejecutar.py
¡Hola!
```

CONQUER
BLOCKS

**CONQUER
BLOCKS**

PYTHON

PROGRAMACION ORIENTEADA
A OBJETOS (POO)

CONQUERBLOCKS



MIEMBROS PRIVADOS (private members)

EJEMPLO: Tenemos un coche que vendemos a 60000 pero cuyo coste de producción es de 10000. Esto último no queríamos que el usuario lo sepa.

```
class Coche:  
    def __init__(self, marca, modelo, anio):  
        """Inicializamos atributos del coche"""  
        self.marca = marca  
        self.modelo = modelo  
        self.anio = anio  
        self.coste_produccion = 10000  
  
    mi_coche = Coche("audi", "a4", 2018)  
    print(mi_coche.coste_produccion)  
| ✓ 0.0s  
10000
```



MIEMBROS PRIVADOS (private members)

EJEMPLO: Tenemos un coche que viene de 10000. Esto último no queríamos

```
class Coche:
    def __init__(self, marca, modelo, anio):
        """Inicializamos atributos"""
        self.marca = marca
        self.modelo = modelo
        self.anio = anio
        self.__coste_produccion = 10000

    mi_coche = Coche("audi", "a4", 2018)
    print(mi_coche.coste_produccion)

✓ 0.0s
10000
```

```
class Coche:
    def __init__(self, marca, modelo, anio):
        """Inicializamos atributos del coche"""
        self.marca = marca
        self.modelo = modelo
        self.anio = anio
        self.__coste_produccion = 10000

    mi_coche = Coche("audi", "a4", 2018)
    print(mi_coche.__coste_produccion)

⊗ 0.1s
-----
AttributeError Traceback (most recent call last)
Cell In[2], line 10
      7     self.__coste_produccion = 10000
      8 mi_coche = Coche("audi", "a4", 2018)
--> 10 print(mi_coche.__coste_produccion)

AttributeError: 'Coche' object has no attribute '__coste_produccion'
```



MIEMBROS PRIVADOS (private members)

EJEMPLO: Tenemos un coche que viene de 10000. Esto último no queríamos

```
class Coche:
    def __init__(self, marca, modelo, anio):
        """Inicializamos atributos"""
        self.marca = marca
        self.modelo = modelo
        self.anio = anio
        self.__coste_produccion = 10000

    mi_coche = Coche("audi", "a4", 2018)
    print(mi_coche.coste_produccion)

✓ 0.0s
10000
```

```
class Coche:
    def __init__(self, marca, modelo, anio):
        """Inicializamos atributos del coche"""
        self.marca = marca
        self.modelo = modelo
        self.anio = anio
        self.__coste_produccion = 10000 Miembro privado

    mi_coche = Coche("audi", "a4", 2018)
    print(mi_coche.__coste_produccion)

⊗ 0.1s
-----
AttributeError Traceback (most recent call last)
Cell In[2], line 10
      7     self.__coste_produccion = 10000
      8 mi_coche = Coche("audi", "a4", 2018)
--> 10 print(mi_coche.__coste_produccion)

AttributeError: 'Coche' object has no attribute '__coste_produccion'
```



MIEMBROS PRIVADOS (private members)

En realidad esto no hace el atributo del todo privado...

```
class Coche:
    def __init__(self, marca, modelo, anio):
        """Inicializamos atributos del coche"""
        self.marca = marca
        self.modelo = modelo
        self.anio = anio
        self.__coste_produccion = 10000  Miembro privado

mi_coche = Coche("audi", "a4", 2018)
print(mi_coche.__coste_produccion)

0.1s

-----
AttributeError                                     Traceback (most recent call last)
Cell In[2], line 10
      7         self.__coste_produccion = 10000
      8 mi_coche = Coche("audi", "a4", 2018)
--> 10 print(mi_coche.__coste_produccion)

AttributeError: 'Coche' object has no attribute '__coste_produccion'
```



MIEMBROS PRIVADOS (private members)

En realidad esto no hace el atributo del todo privado...

```
class Coche:
    def __init__(self, marca, modelo, anio):
        """Inicializamos atributos del coche"""
        self.marca = marca
        self.modelo = modelo
        self.anio = anio
        self.__coste_produccion = 10000

    mi_coche = Coche("audi", "a4", 2018)
    print(mi_coche.__coste_produccion)

    0.0s

10000
```



MIEMBROS PRIVADOS (private members)

```
class Coche:  
    def __init__(self, marca, modelo, anio):  
        """Inicializamos atributos del coche"""  
        self.marca = marca  
        self.modelo = modelo
```

Lo que llamamos atributos privados en python no tiene que ver con privacidad real

```
mi_coche = Coche("audi", "a4", 2018)  
print(mi_coche._Coche__coste_produccion)  
✓ 0.0s  
10000
```



MIEMBROS PRIVADOS (private members)

¿Para qué nos interesan?

1. Encapsulación:

Al marcar un atributo como privado, estás indicando que ese atributo no debe ser accedido directamente desde fuera de la clase.

Esto permite un mejor control sobre cómo los datos son manipulados y protege la integridad de los datos internos de la clase.

```
class Coche:  
    def __init__(self, marca, modelo, anio):  
        """Inicializamos atributos del coche"""  
        self.marca = marca  
        self.modelo = modelo  
        self.anio = anio  
        self.__coste_produccion = 10000  
  
class CocheElectricoo(Coche):  
    """Representa aspectos de un coche,  
    especifico para coches electricos"""  
    def __init__(self, marca, modelo, anio):  
        super().__init__(marca, modelo, anio)  
        self.tamanio_bateria = 70
```



MIEMBROS PRIVADOS (private members)

¿Para qué nos interesan?

1. Encapsulación:

Al marcar un atributo como privado, estás indicando que ese atributo no debe ser accedido directamente desde fuera de la clase.

Esto permite un mejor control sobre cómo los datos son manipulados y protege la integridad de los datos internos de la clase.

```

class Coche:
    def __init__(self, marca, modelo, año):
        self.marca = marca
        self.modelo = modelo
        self.año = año
        self.tamanio_bateria = 70

    def __str__(self):
        return f'{self.marca} {self.modelo} {self.año}'

mi_audi = Coche("audi", "a4", 2015)
mi_tesla = CocheElectrico("tesla", "modelo s", 2016)
print(mi_tesla.__CocheElectrico__coste_produccion)

0.0s
-----
AttributeError Traceback (most recent call last)
Cell In[8], line 19
    17 mi_audi = Coche("audi", "a4", 2015)
    18 mi_tesla = CocheElectrico("tesla", "modelo s", 2016)
--> 19 print(mi_tesla.__CocheElectrico__coste_produccion)
...
esp: AttributeError: 'CocheElectrico' object has no attribute '__CocheElectrico__coste_produccion'

```



MIEMBROS PRIVADOS (private members)

¿Para qué nos interesan?

2. Control de Acceso:

Los atributos privados restringen el acceso directo a los datos desde fuera de la clase.

Solo los métodos dentro de la misma clase pueden acceder y modificar estos atributos. Esto evita modificaciones accidentales o inapropiadas de los datos.

```

class Coche:
    def __init__(self, marca, modelo, año):
        self.marca = marca
        self.modelo = modelo
        self.año = año
        self.tamanio_bateria = 70

    def __str__(self):
        return f'{self.marca} {self.modelo} {self.año}'

mi_audi = Coche("audi", "a4", 2015)
mi_tesla = CocheElectrico("tesla", "modelo s", 2016)
print(mi_tesla.__CocheElectrico__coste_produccion)

0.0s
-----
AttributeError Traceback (most recent call last)
Cell In[8], line 19
    17 mi_audi = Coche("audi", "a4", 2015)
    18 mi_tesla = CocheElectrico("tesla", "modelo s", 2016)
--> 19 print(mi_tesla.__CocheElectrico__coste_produccion)
...
esp: AttributeError: 'CocheElectrico' object has no attribute '__CocheElectrico__coste_produccion'

```



MIEMBROS PRIVADOS (private members)

¿Para qué nos interesan?

3. Evitar Colisiones

Al usar atributos privados, reduce el riesgo de colisiones de nombres con atributos de otras clases o del código externo

```

class Coche:
    def __init__(self, marca, modelo, año):
        self.marca = marca
        self.modelo = modelo
        self.año = año
        self.tamanio_bateria = 70

    def __str__(self):
        return f'{self.marca} {self.modelo} {self.año}'

mi_audi = Coche("audi", "a4", 2015)
mi_tesla = CocheElectrico("tesla", "modelo s", 2016)
print(mi_tesla._CocheElectrico__coste_produccion)

0.0s

-----
AttributeError Traceback (most recent call last)
Cell In[8], line 19
      17 mi_audi = Coche("audi", "a4", 2015)
      18 mi_tesla = CocheElectrico("tesla", "modelo s", 2016)
--> 19 print(mi_tesla._CocheElectrico__coste_produccion)
      ...
      espi AttributeError: 'CocheElectrico' object has no attribute '_CocheElectrico__coste_produccion'

super().__init__(marca, modelo, año)
self.tamanio_bateria = 70

```



MIEMBROS PRIVADOS (private members)

¿Para qué nos interesan?

4. Documentación y Abstracción

Al marcar los atributos como privados, estás señalando a otros programadores que estos atributos no están destinados a ser accedidos directamente y que deben consultar la documentación de la clase para comprender cómo interactuar adecuadamente con ella.

```

class Coche:
    def __init__(self, marca, modelo, año):
        self.marca = marca
        self.modelo = modelo
        self.año = año
        self.tamanio_bateria = 70

    def __str__(self):
        return f'{self.marca} {self.modelo} {self.año}'

mi_audi = Coche("audi", "a4", 2015)
mi_tesla = CocheElectrico("tesla", "modelo s", 2016)
print(mi_tesla._CocheElectrico__coste_produccion)

0.0s

-----
AttributeError Traceback (most recent call last)
Cell In[8], line 19
      17 mi_audi = Coche("audi", "a4", 2015)
      18 mi_tesla = CocheElectrico("tesla", "modelo s", 2016)
--> 19 print(mi_tesla._CocheElectrico__coste_produccion)
      ...
      espi AttributeError: 'CocheElectrico' object has no attribute '_CocheElectrico__coste_produccion'

super().__init__(marca, modelo, año)
self.tamanio_bateria = 70

```



MIEMBROS PRIVADOS (private members)

Podemos aplicar esto mismo a los métodos...

```
class Coche:
    def __init__(self, marca, modelo, anio):
        """Inicializamos atributos del coche"""
        self.marca = marca
        self.modelo = modelo
        self.anio = anio
        self.__coste_produccion = 10000

    def __secreto(self):
        print("Es metodo es privado")

mi_coche = Coche("audi", "a4", 2015)
mi_coche.__secreto()
```



MIEMBROS PRIVADOS (private members)

Podemos aplicar esto mismo a los métodos...

```
class Coche:
    def __init__(self, marca, modelo, anio):
        """Inicializamos atributos del coche"""
        self.marca = marca
        self.modelo = modelo
        self.anio = anio
        self.__coste_produccion = 10000

    def __secreto(self):
        print("Es metodo es privado")

mi_coche = Coche("audi", "a4", 2015)
mi_coche.__secreto()
```

AttributeError
Cell In[9], line 13
10 print("Es metodo es privado")
11 mi_coche = Coche("audi", "a4", 2015)
12 mi_coche.__secreto()
--> 13 mi_coche.__secreto()
Se
Se AttributeError: 'Coche' object has no attribute '__secreto'
Se
Se
self.__coste_produccion = 10000



MIEMBROS PRIVADOS (private members)

Podemos aplicar esto mismo a los métodos...

The screenshot shows two code cells. The left cell contains:

```

class Coche:
    def __init__(self, marca, modelo, anio):
        self.marca = marca
        self.modelo = modelo
        self.anio = anio
        self.__coste_produccion = 10000

    def __secreto(self):
        print("Es metodo es privado")

mi_coche = Coche("audi", "a4", 2015)
mi_coche.__secreto()

```

The right cell contains:

```

class Coche:
    def __init__(self, marca, modelo, anio):
        """Inicializamos atributos del coche"""
        self.marca = marca
        self.modelo = modelo
        self.anio = anio
        self.__coste_produccion = 10000

    def __secreto(self):
        print("Es metodo es privado")

mi_coche = Coche("audi", "a4", 2015)
mi_coche._Coche__secreto()

```

The output of the right cell is:

```

✓ 0.0s
Es metodo es privado

```



IMPORTACION DE CLASES

IMPORTACION DE UNA CLASE:

The interface shows a file tree on the right with:

- importacion_clases
 - > __pycache__
 - coche.py
 - mi_coche.py

The left pane shows the content of `coche.py`:

```

class Coche:
    def __init__(self, marca, modelo, anio):
        """Inicializamos atributos del coche"""
        self.marca = marca
        self.modelo = modelo
        self.anio = anio
        self.__coste_produccion = 10000

    def llenar_deposito(self):
        """Simula el llenado del deposito del coche"""
        self.deposito = 100
        print("El deposito esta a", self.deposito)

```

The right pane shows the code in `mi_coche.py` and its execution in the terminal:

```

importacion_clases > mi_coche.py > [0] anio
1   from coche import Coche
2
3   mi_nuevo_coche = Coche("audi", "a4", 2016)
4   mi_nuevo_coche.llenar_deposito()
5
6   mi_nuevo_coche.anio = 2014
7   print(mi_nuevo_coche.anio)

PROBLEMAS SALIDA CONSOLA DE DEPURACIÓN TERMINAL
● (cblocks) MacBook-Pro-5:Clase 9 Elena$ /Users/Elena/miniconda3/envs/Avanzado/Clase 9/importacion_clases/mi_coche.py"
El deposito esta a 100
2014
○ (cblocks) MacBook-Pro-5:Clase 9 Elena$ []

```



IMPORTACION DE CLASES

GUARDAR MÚLTIPLES CLASES EN UN MÓDULO:

```

mi_coche_elec.py X

importacion_clases > mi_coche_elec.py > ...
1   from coche import CocheElectrico
2
3   mi_tesla = CocheElectrico("tesla", "modelo s", 2016)
4
5   print(mi_tesla.bateria.tamano_bateria)

PROBLEMAS SALIDA CONSOLA DE DEPURACIÓN TERMINAL JUPYTER

● (cblocks) MacBook-Pro-5:Clase 9 Elena$ /Users/Elena/miniconda3/python/Avanzado/Clase 9/importacion_clases/mi_coche_elec.py"
70
○ (cblocks) MacBook-Pro-5:Clase 9 Elena$ █

```

```

class Coche:
    def __init__(self, marca, modelo, anio):
        """Inicializamos atributos del coche"""
        self.marca = marca
        self.modelo = modelo
        self.anio = anio
        self.deposito = 0

    def llenar_deposito(self):
        """Simula el llenado del deposito del coche"""
        self.deposito = 100
        print("El deposito esta a", self.deposito)

class Bateria:
    """Un intento simple de modelizar
    una bateria"""

    def __init__(self, tamano_bateria = 70):
        """Inicializamos los atributos de la bateria"""
        self.tamano_bateria = tamano_bateria

    def describir_bateria(self):
        """Imprimir el tamaño de la bateria"""
        print("Este coche tiene una bateria de tamaño",
              self.tamano_bateria, "kWh.")

class CocheElectrico(Coche):
    """Representa aspectos de un coche,
    especifico para coches electricos"""
    def __init__(self, marca, modelo, anio):
        """ Inicializar atributos de superclase.
        Despues inicializar atributos especificos
        del coche electrico """
        super().__init__(marca, modelo, anio)
        self.bateria = Bateria()

mi_tesla = CocheElectrico("tesla", "modelo s", 2016)
mi_tesla.llenar_deposito()
mi_tesla.bateria.describir_bateria()

```



IMPORTACION DE CLASES

IMPORTAR UNA PARENT CLASS:

```

coche.py X

importacion_clases > coche.py > ...
1   class Coche:
2       def __init__(self, marca, modelo, anio):
3           """Inicializamos atributos del coche"""
4           self.marca = marca
5           self.modelo = modelo
6           self.anio = anio
7           self.__coste_produccion = 10000
8
9       def llenar_deposito(self):
10          """Simula el llenado del deposito del coche"""
11          self.deposito = 100
12          print("El deposito esta a", self.deposito)
13

```

```

coche_electrico.py X

importacion_clases > coche_electrico.py > ...
1   from coche import Coche
2
3   class Bateria:
4       """Un intento simple de modelizar
una bateria"""
5
6   def __init__(self, tamano_bateria = 70):
7       """Inicializamos los atributos de la bateria"""
8       self.tamano_bateria = tamano_bateria
9
10  def describir_bateria(self):
11      """Imprimir el tamaño de la bateria"""
12      print("Este coche tiene una bateria de tamaño",
13            self.tamano_bateria, "kWh.")
14
15  class CocheElectrico(Coche):
16      """Representa aspectos de un coche,
especifico para coches electricos"""
17      def __init__(self, marca, modelo, anio):
18          """ Inicializar atributos de superclase.
Despues inicializar atributos especificos
del coche electrico """
19          super().__init__(marca, modelo, anio)
20          self.bateria = Bateria()
21
22
23
24

```



GUIA DE ESTILO...

- **Variables:**

- ▶ Letras minúsculas
- ▶ Separamos las palabras con guiones bajos (snake_case)
- ▶ Nombres descriptivos que transmitan el propósito de la variable

- **Funciones:**

- ▶ Letras minúsculas
- ▶ Separamos las palabras con guiones bajos (snake_case)
- ▶ Usa verbos para describir las acciones

- **Constantes:**

- ▶ Letras mayúsculas
- ▶ Separamos las palabras con guiones bajos (SNAKE_CASE)
- ▶ Nombres claros y concisos que transmitan el propósito de la constante

- **Clases:**

- ▶ Usa CamelCase (capitaliza la primera letra de cada palabra sin espacios)
- ▶ Nombres descriptivos que reflejen la naturaleza de la clase



GUIA DE ESTILO...

- ▶ Los nombres de instancias y módulos se escriben en snake_case

```
mi_instancia = MiClaseEjemplo()
otra_instancia = OtraClaseEjemplo()

import modulo_personalizado
import modulo_estandar
```



GUIA DE ESTILO...

- ▶ Cada clase debe tener una cadena de documentación (docstring) inmediatamente después de la definición de la clase. La cadena de documentación debe ser una breve descripción de lo que hace la clase, y debes seguir las mismas convenciones de formato que usaste para escribir cadenas de documentación en funciones.

```
"""
Módulo personalizado para realizar tareas específicas.
Contiene varias clases que pueden ser útiles en diversos escenarios.

"""

class MiClase:
    """Esta es una clase de ejemplo que muestra cómo usar docstrings."""
    def __init__(self):
        pass
```



GUIA DE ESTILO...

- ▶ Cada módulo también debe tener una cadena de documentación que describa para qué se pueden usar las clases en un módulo.

```
"""
Módulo personalizado para realizar tareas específicas.
Contiene varias clases que pueden ser útiles en diversos escenarios.

"""

class MiClase:
    """Esta es una clase de ejemplo que muestra cómo usar docstrings."""
    def __init__(self):
        pass
```



GUIA DE ESTILO...

- ▶ Puedes usar líneas en blanco para organizar el código, pero no las uses en exceso. Dentro de una clase, puedes usar una línea en blanco entre métodos, y dentro de un módulo, puedes usar dos líneas en blanco para separar clases.

```
class MiClase:  
    def __init__(self):  
        pass  
  
    def metodo_uno(self):  
        pass  
  
class OtraClase:  
    def __init__(self):  
        pass  
  
    def metodo_dos(self):  
        pass
```



GUIA DE ESTILO...

- ▶ Si necesitas importar un módulo de la biblioteca estándar y un módulo que escribiste, coloca la declaración de importación para el módulo de la biblioteca estándar primero. Luego agrega una línea en blanco y la declaración de importación para el módulo que escribiste. En programas con múltiples declaraciones de importación, esta convención facilita ver de dónde provienen los diferentes módulos utilizados en el programa.

```
import os  
  
import mi_modulo_personalizado  
  
# Resto del código del programa
```

```
# standard modules  
import os  
  
# other modules  
import mi_modulo_personalizado  
  
# Resto del código del programa
```

CONQUER
BLOCKS

<!--SOLID-->

Que es la programación orientada a objetos? {

<Por="Raquel Martinez"/>

}

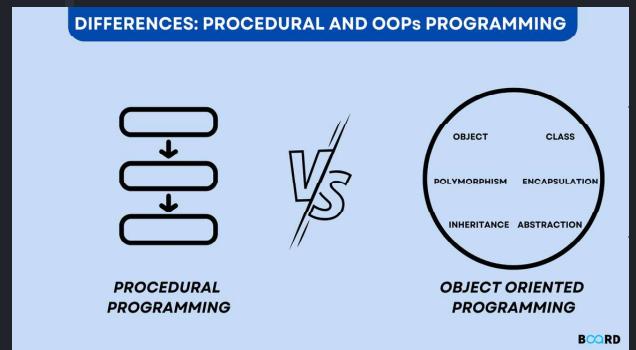


Contenidos

- 01 Paradigma
- 02 Arquitectura de una clase
- 03 Objetos
- 04 Encapsulación
- 05 Abstracción
- 06 Herencia
- 07

Paradigma {

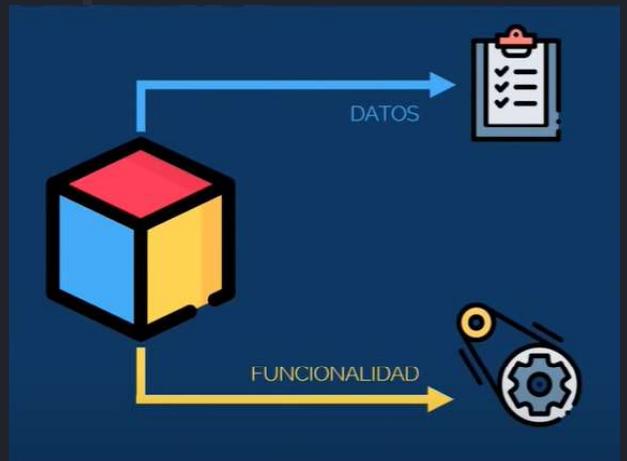
- Basado en el concepto de "objetos"
- Objetos de la vida real
- Objeto -> datos y comportamientos



}

Paradigma {

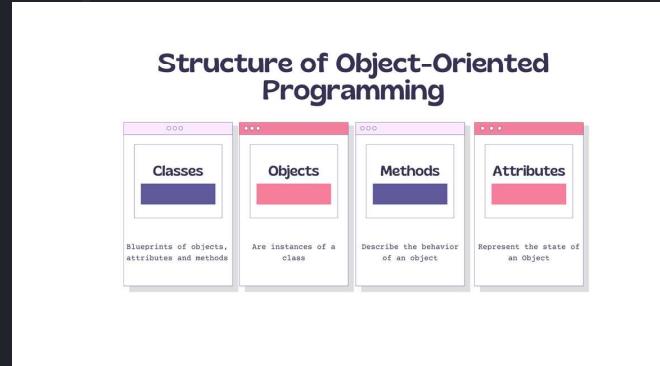
- Basado en el concepto de "objetos"
- Objetos de la vida real
- Objeto -> datos y comportamientos



}

Clases{

- Atributos
- Métodos de instancia
- Métodos de clase
- Métodos estáticos



}

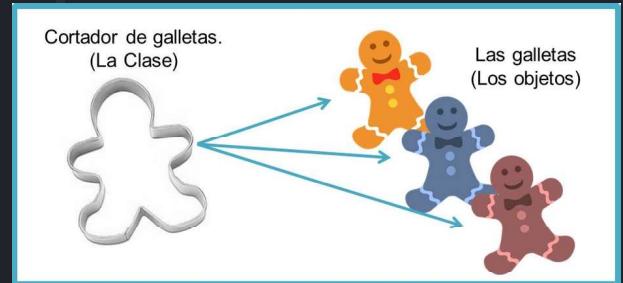
Instancia y objeto {

Objeto:

- Un objeto es una entidad que representa una instancia concreta de una clase. Métodos de instancia

Instancia:

- Una instancia es cada objeto individual creado a partir de una clase.



}

```
<!--SOLID-->
```

Codifiquemos ! {

```
<Ejemplo/>
```

```
}
```

```
Abstracción {
```

```
Captar características
```

```
Separar características
```

Una abstracción se enfoca en la visión externa de un objeto, separa el comportamiento específico de un objeto

```
}
```

Encapsulación {

Atributos públicos

Accesibles por instancias

Atributos privados

Los atributos privados se definen con un doble guión bajo (__) al comienzo de su nombre.

Atributos protegidos

Indica que el atributo no debe ser accedido directamente desde fuera de la clase, pero no impide su acceso.

Principio de ocultar los detalles internos de un objeto y controlar el acceso a sus atributos y métodos

}

Herencia

Reutilización de código

Promueve la reutilización del código y evitando la duplicación de funcionalidades.

Estructura jerárquica

Clases más específicas (subclases) pueden heredar características de clases más generales (superclases).

La herencia permite que una clase hija herede atributos y métodos de una clase padre. Esto promueve la reutilización de código y facilita la creación de jerarquías de clases.

Abstracción de comportamiento común

La herencia facilita la creación de clases que encapsulan comportamientos comunes

}

Polimorfismo

Flexibilidad en el código

El polimorfismo permite escribir código más flexible y genérico al tratar objetos de diferentes clases de manera uniforme si comparten una interfaz común.

Facilita la extensión del software

nuevas clases pueden implementar la misma interfaz que las clases existentes y ser utilizadas en el mismo contexto.

El polimorfismo es un concepto de la programación orientada a objetos que se refiere a la capacidad de diferentes objetos de responder de manera diferente a la misma invocación de método.

}

<!--SOLID-->

Gracias {

<Por="Raquel Martinez/>

}

```
<!--SOLID-->
```

Introducción a SOLID

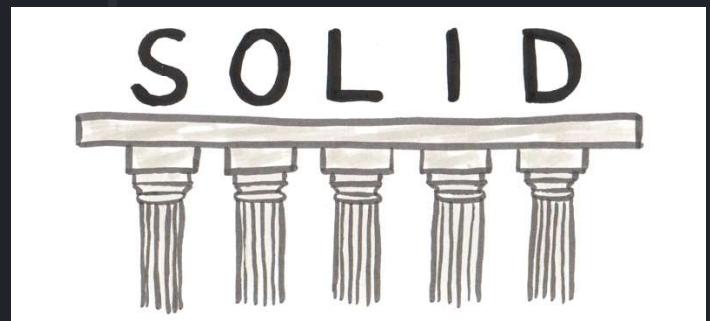
```
<Por="Raquel Martinez"/>
```

```
}
```



```
SOLID {
```

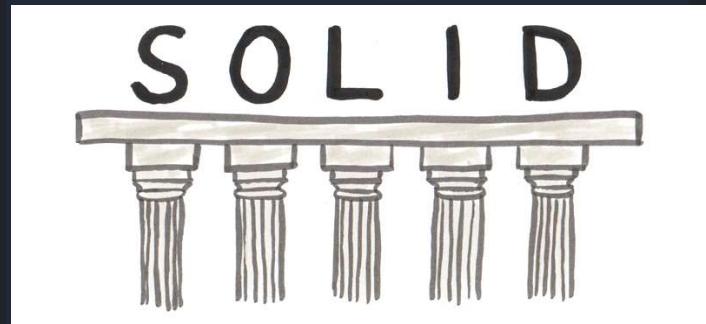
Son un conjunto de reglas y mejores prácticas a seguir al diseñar una estructura de clase.



```
}
```

SOLID {

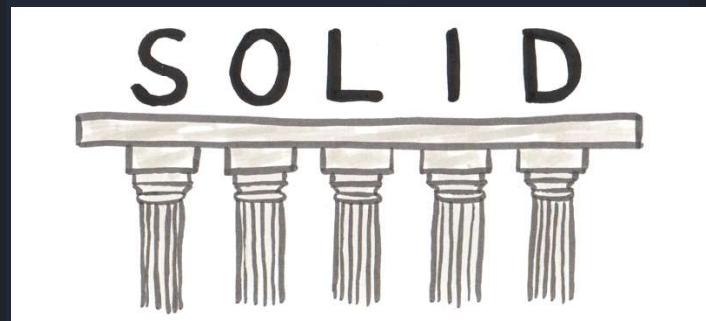
- Los principios SOLID fueron introducidos por primera vez por el famoso científico informático Robert J. Martin



}

SOLID {

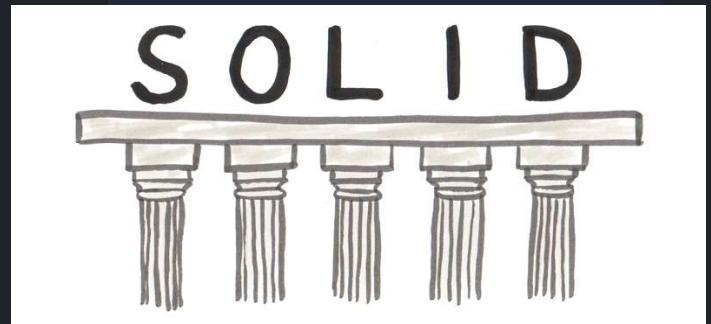
"Para crear código comprensible, legible y comprobable en el que muchos desarrolladores puedan trabajar en colaboración."



}

SOLID {

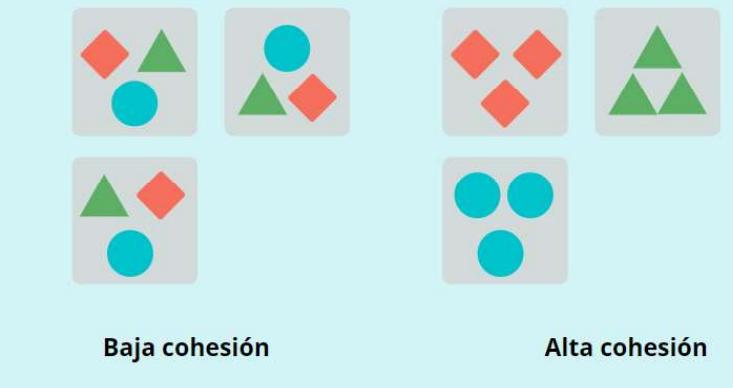
- 1. Cohesión
- 2. Acoplamiento



}

Cohesión {

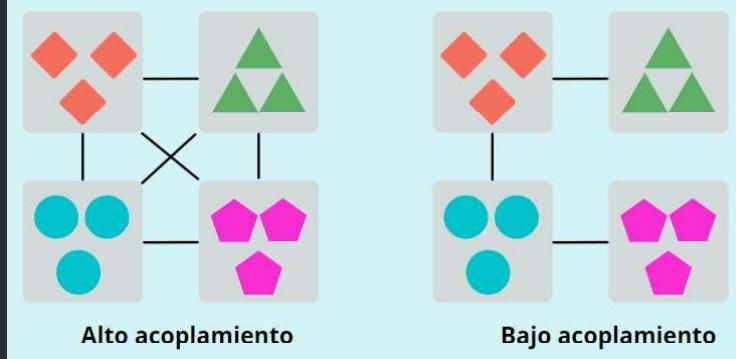
- La cohesión se refiere a la medida en que las responsabilidades y funciones de un módulo (clase, método o componente) están relacionadas y enfocadas en una única tarea o propósito.



}

Acoplamiento {

- Grado de interdependencia entre los diferentes módulos de un sistema.

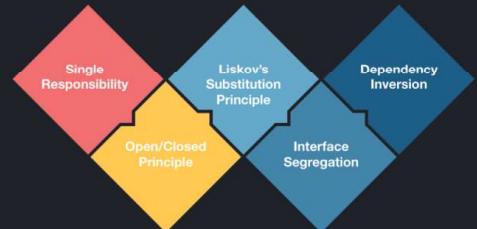


}

Single Responsibility Principle - Principio de responsabilidad única {

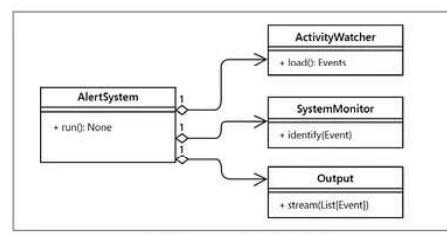
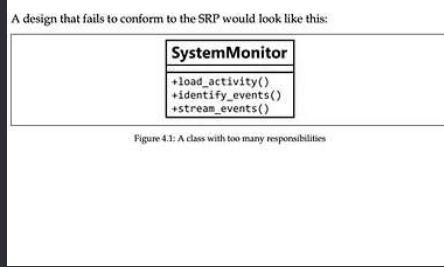
- Una clase debe hacer una cosa y, por lo tanto, debe tener una sola razón para cambiar.

S.O.L.I.D.



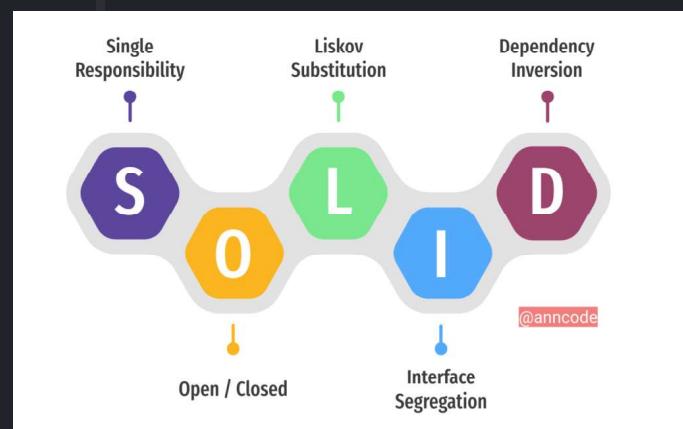
}

Single Responsibility Principle - Principio de responsabilidad única {



S DE SOLID {

- Claridad de la responsabilidad
- Facilita el mantenimiento
- Promueve la cohesión
- Facilita la reutilización del código



{}

SOLID {

```
public class Factura {  
  
    private Libro libro;  
    private int cantidad;  
    private double tasaDescuento;  
    private double tasaImpuesto;  
    private double total;  
  
    public Factura(Libro libro, int cantidad, double tasaDescuento, double tasaImpuesto) {  
        this.libro = libro;  
        this.cantidad = cantidad;  
        this.tasaDescuento = tasaDescuento;  
        this.tasaImpuesto = tasaImpuesto;  
        this.total = this.calculaTotal();  
    }  
  
    public double calculaTotal() {  
        double precio = ((libro.precio - libro.precio * tasaDescuento) * this.cantidad);  
  
        double precioConImpuestos = precio * (1 + tasaImpuesto);  
  
        return precioConImpuestos;  
    }  
  
    public void imprimeFactura() {  
        System.out.println(cantidad + "x " + libro.nombre + " " + libro.precio + "$");  
        System.out.println("Tasa de Descuento: " + tasaDescuento);  
        System.out.println("Tasa de Impuesto: " + tasaImpuesto);  
        System.out.println("Total: " + total);  
    }  
  
    public void guardarArchivo(String nombreArchivo) {  
        // Crea un archivo con el nombre dado y escribe la factura.  
    }  
}
```

}

SOLID {

```
public class FacturaImpresion {  
    private Factura factura;  
  
    public FacturaImpresion(Factura factura) {  
        this.factura = factura;  
    }  
  
    public void imprimir() {  
        System.out.println(factura.cantidad + "x " + factura.libro.nombre + " " + factura.libro.precio + "$");  
        System.out.println("Tasa de Descuento: " + factura.tasaDescuento);  
        System.out.println("Tasa de Impuesto: " + factura.tasaImpuesto);  
        System.out.println("Total: " + factura.total + " $");  
    }  
}
```

}

SOLID {

```
public class FacturaPersistencia {  
    Factura factura;  
  
    public FacturaPersistencia(Factura factura) {  
        this.factura = factura;  
    }  
  
    public void guardarArchivo(String nombreArchivo) {  
        // Crea un archivo con el nombre dado y escribe la factura.  
    }  
}
```

}

<!--SOLID-->

Codifiquemos! {

<Ejemplo/>

}

```
<!--SOLID-->
```

Gracias {

```
<Por="Raquel Martinez/>
```

```
}
```

```
<!--SOLID-->
```

SOLID - Principio abierto/cerrado

```
<Por="Raquel Martinez"/>
```

```
}
```



```
SOLID {
```



```
}
```

Definición{

Un artefacto de software debe estar abierto a ampliaciones pero cerrado a modificaciones.



}

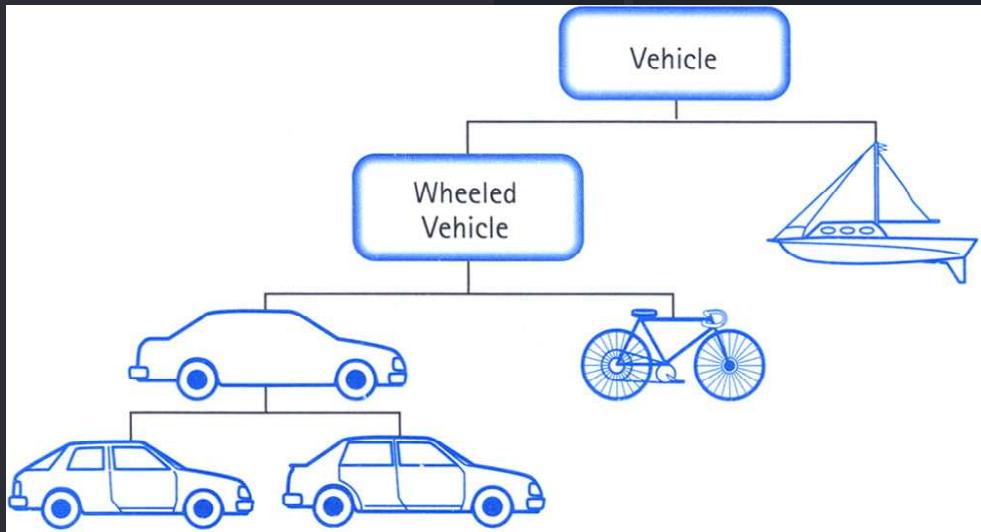
Beneficios {

- Facilita el mantenimiento y la evolución del código a largo plazo.
- Mejora la modularidad y la reutilización del código.
- Reduce el riesgo de introducir errores al modificar el código existente.
- Promueve un diseño más flexible y extensible.

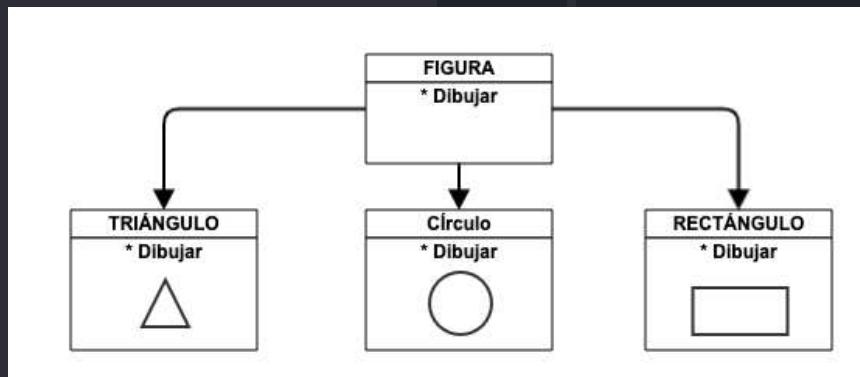


}

Implementación en Python - Herencia {

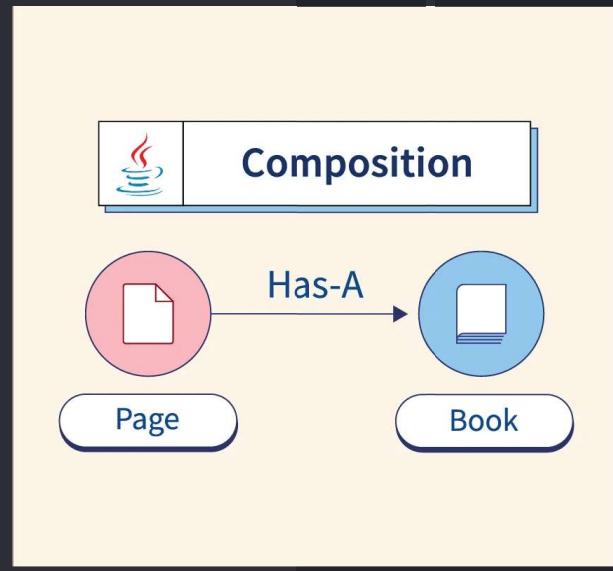


Implementación en Python - Polimorfismo {



}

Implementación en Python - Composición{



WO - OC {

```
from math import pi

class Shape:
    def __init__(self, shape_type, **kwargs):
        self.shape_type = shape_type
        if self.shape_type == "rectangle":
            self.width = kwargs["width"]
            self.height = kwargs["height"]
        elif self.shape_type == "circle":
            self.radius = kwargs["radius"]

    def calculate_area(self):
        if self.shape_type == "rectangle":
            return self.width * self.height
        elif self.shape_type == "circle":
            return pi * self.radius**2
```

}

```

from abc import ABC, abstractmethod
from math import pi

W - OC {

    class Shape(ABC):
        def __init__(self, shape_type):
            self.shape_type = shape_type

        @abstractmethod
        def calculate_area(self):
            pass

    class Circle(Shape):
        def __init__(self, radius):
            super().__init__("circle")
            self.radius = radius

        def calculate_area(self):
            return pi * self.radius**2

    class Rectangle(Shape):
        def __init__(self, width, height):
            super().__init__("rectangle")
            self.width = width
            self.height = height

        def calculate_area(self):
            return self.width * self.height

    class Square(Shape):
        def __init__(self, side):
            super().__init__("square")
            self.side = side

        def calculate_area(self):
            return self.side**2d
}

```

Implementación de diferentes algoritmos de clasificación {

```

from abc import ABC, abstractmethod

class Classifier(ABC):
    @abstractmethod
    def train(self, data, labels):
        pass

    @abstractmethod
    def predict(self, data):
        pass
}

```

Implementación de diferentes algoritmos de clasificación {

```
class NaiveBayesClassifier(Classifier):
    def train(self, data, labels):
        # Implementación del entrenamiento de Naive Bayes

    def predict(self, data):
        # Implementación de la predicción de Naive Bayes

class NeuralNetworkClassifier(Classifier):
    def train(self, data, labels):
        # Implementación del entrenamiento de Red Neuronal

    def predict(self, data):
        # Implementación de la predicción de Red Neuronal

class SVMClassifier(Classifier):
    def train(self, data, labels):
        # Implementación del entrenamiento de SVM

    def predict(self, data):
        # Implementación de la predicción de SVM
```

}

Implementación de diferentes algoritmos de enrutamiento {

```
from abc import ABC, abstractmethod

class RoutingAlgorithm(ABC):
    @abstractmethod
    def calculate_routes(self, network_topology):
        pass

    @abstractmethod
    def update_routes(self, network_event):
        pass
```

}

Implementación de diferentes algoritmos de enrutamiento {

```
class DistanceVectorRouting(RoutingAlgorithm):
    def calculate_routes(self, network_topology):
        # Implementación del cálculo de rutas de vector de distancia

    def update_routes(self, network_event):
        # Implementación de la actualización de rutas de vector de distancia

class LinkStateRouting(RoutingAlgorithm):
    def calculate_routes(self, network_topology):
        # Implementación del cálculo de rutas de estado de enlace

    def update_routes(self, network_event):
        # Implementación de la actualización de rutas de estado de enlace

class PolicyBasedRouting(RoutingAlgorithm):
    def calculate_routes(self, network_topology):
        # Implementación del cálculo de rutas basado en políticas

    def update_routes(self, network_event):
        # Implementación de la actualización de rutas basada en políticas }
```

<!--SOLID-->

Codifiquemos! {

<Ejemplo/>

}

```
<!--SOLID-->
```

Gracias {

```
<Por="Raquel Martinez/>
```

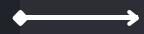
```
}
```

<!--SOLID-->

SOLID - Principio de Sustitución de Liskov (LSP)

<Por="Raquel Martinez"/>

}



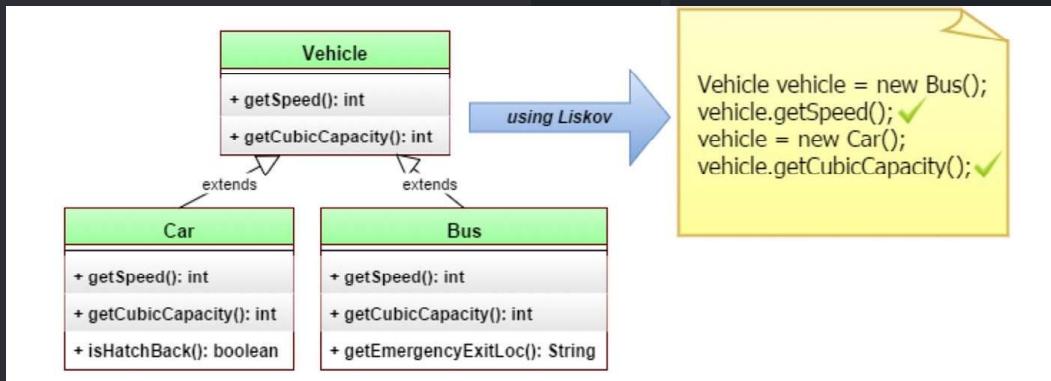
Definición{

Si S es un subtipo de T, entonces los objetos de tipo T en un programa pueden ser reemplazados por objetos de tipo S sin alterar ninguna de las propiedades deseables del programa (correctitud, tarea realizada, etc.).

}

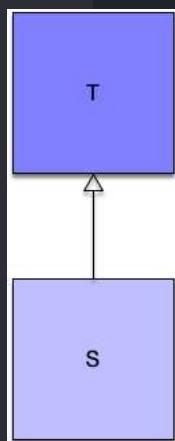
Definición{

Si S es un subtipo de T, entonces los objetos de tipo T en un programa pueden ser reemplazados por objetos de tipo S sin alterar ninguna de las propiedades deseables del programa (correctitud, tarea realizada, etc.).



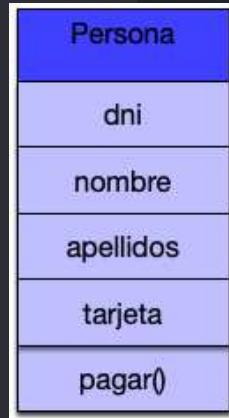
Beneficios {

- Reutilización del código
- Mantenimiento del código
- Polimorfismo
- Correctitud



}

Herencia{



}

Herencia{



}

```
package com.arquitecturajava;

public class Niño extends Persona{

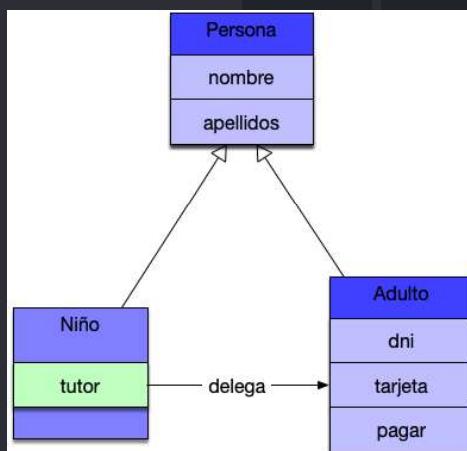
    public Niño(String nombre, String apellidos) {
        super(null, nombre, apellidos, null);
        // TODO Auto-generated constructor stub
    }

    @Override
    public void pagar() {
        // TODO Auto-generated method stub
        throw new RuntimeException("un niño no puede pagar");
    }

}
```

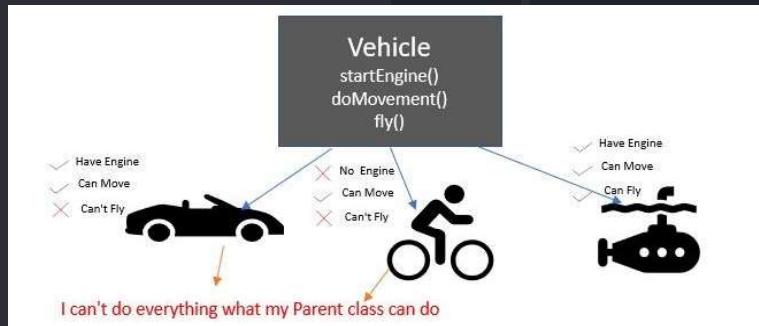
}

Herencia{



}

Herencia{



Herencia{

```
public class DeviceWithoutEngine extends TransportationDevice {  
    public void startMoving(){  
        System.out.println("Basic functionality of moving for device  
without engine");  
    }  
}  
  
public class DeviceWithEngine {  
    public void startEngine(){  
        System.out.println("Basic engine start functionality");  
    }  
}
```

}

<!--SOLID-->

Codifiquemos ! {

<Ejemplo/>

}

<!--SOLID-->

Gracias {

<Por="Raquel Martinez/>

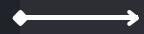
}

```
<!--SOLID-->
```

SOLID - Segregación de interfaz

```
<Por="Raquel Martinez"/>
```

```
}
```



```
Definición{
```

No se debe obligar a los clientes a depender de métodos que no utilizan. Las interfaces pertenecen a clientes, no a jerarquías.

```
}
```

Definición{

Si una clase no utiliza métodos o atributos particulares, entonces esos métodos y atributos deben segregarse en clases más específicas.

}

Interfaz {

En la programación orientada a objetos, un interfaz define al conjunto de métodos que tiene que tener un objeto para que pueda cumplir una determinada función en nuestro sistema. Dicho de otra manera, un interfaz define como se comporta un objeto y lo que se puede hacer con él.

}

Interfaz informal {

```
class Mando:  
    def siguiente_canal(self):  
        pass  
    def canal_anterior(self):  
        pass  
    def subir_volumen(self):  
        pass  
    def bajar_volumen(self):  
        pass
```

```
class MandoLG(Mando):  
    def siguiente_canal(self):  
        print("LG->Siguiente")  
    def canal_anterior(self):  
        print("LG->Anterior")  
    def subir_volumen(self):  
        print("LG->Subir")  
    def bajar_volumen(self):  
        print("LG->Bajar")
```

}

Interfaz formal {

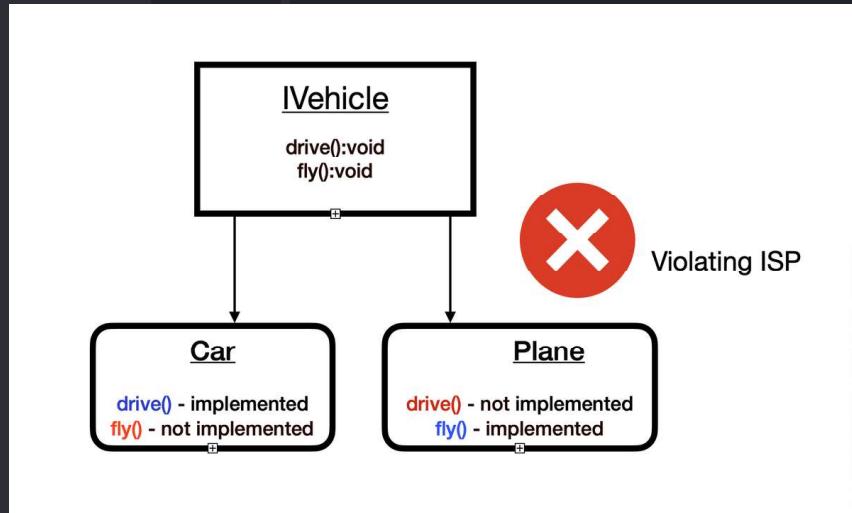
```
from abc import abstractmethod  
from abc import ABCMeta  
  
class Mando(metaclass=ABCMeta):  
    @abstractmethod  
    def siguiente_canal(self):  
        pass  
  
    @abstractmethod  
    def canal_anterior(self):  
        pass  
  
    @abstractmethod  
    def subir_volumen(self):  
        pass  
  
    @abstractmethod  
    def bajar_volumen(self):  
        pass
```

```
class MandoSamsung(Mando):  
    def siguiente_canal(self):  
        print("Samsung->Siguiente")  
    def canal_anterior(self):  
        print("Samsung->Anterior")  
    def subir_volumen(self):  
        print("Samsung->Subir")  
    def bajar_volumen(self):  
        print("Samsung->Bajar")
```

}

Beneficios {

- Mantenimiento
- Flexibilidad
- Reutilización



}

Ejemplo {

```
from abc import ABC, abstractmethod

class Worker(ABC):
    @abstractmethod
    def work(self):
        pass

    @abstractmethod
    def eat(self):
        pass
```

}

Ejemplo {

```
class HumanWorker(Worker):
    def work(self):
        print("Human is working")

    def eat(self):
        print("Human is eating")

class RobotWorker(Worker):
    def work(self):
        print("Robot is working")

    def eat(self):
        raise NotImplementedError("Robots do not eat")
```

}

Ejemplo {

```
from abc import ABC, abstractmethod

class Workable(ABC):
    @abstractmethod
    def work(self):
        pass

class Eatble(ABC):
    @abstractmethod
    def eat(self):
        pass
```

}

Ejemplo {

```
class HumanWorker(Workable, Eatable):
    def work(self):
        print("Human is working")

    def eat(self):
        print("Human is eating")

class RobotWorker(Workable):
    def work(self):
        print("Robot is working")
```

}

Protocol {

- Un protocolo es un conjunto de métodos o atributos que un objeto debe tener para ser considerado compatible con ese protocolo. Los protocolos le permiten definir interfaces sin crear explícitamente una clase o heredar de una clase base específica.

typing.Protocol

}

<!--SOLID-->

Codifiquemos ! {

<Ejemplo/>

}

<!--SOLID-->

Gracias {

<Por="Raquel Martinez/>

}

<!--SOLID-->

SOLID - Principio de Inversión de Dependencias

<Por="Raquel Martinez"/>

}



Definición{

- Los módulos de alto nivel no deberían depender de los módulos de bajo nivel. Ambos deberían depender de abstracciones.
- Las abstracciones no deberían depender de los detalles. Los detalles deberían depender de las abstracciones.

}

Concepto - Módulos de Alto Nivel {

```
class App:  
    def __init__(self, notificador: Notificador):  
        self.notificador = notificador  
  
    def enviar_notificacion(self, mensaje: str):  
        self.notificador.enviar(mensaje)  
        print("Notificación enviada correctamente.")
```

}

Concepto - Módulos de Bajo Nivel {

```
class EmailNotificador(Notificador):  
    def enviar(self, mensaje: str):  
        print(f'Enviando email: {mensaje}')
```

}

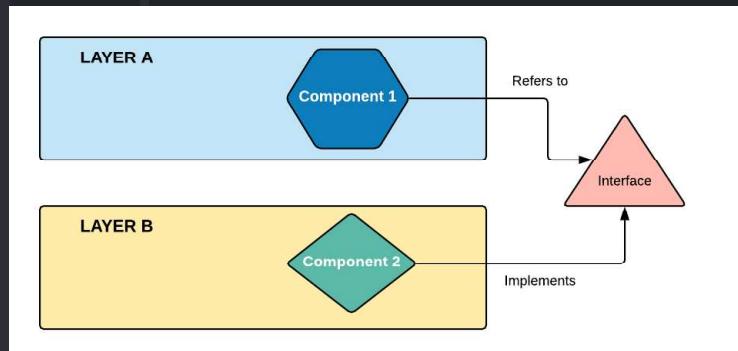
Concepto - Abstracciones {

- Estas son interfaces o clases abstractas que definen los métodos que los módulos de alto y bajo nivel deben implementar.
- Ejemplo: una interfaz Notificador que declara un método enviar(mensaje: str).

}

Beneficios {

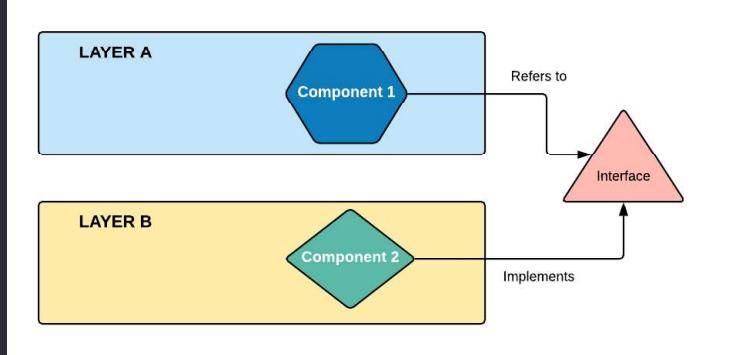
- Reducción del Acoplamiento
- Mejora de la Modularidad
- Facilita el Testing
- Mantenibilidad y Extensibilidad



}

Comparación con Otros Principios SOLID {

- Responsabilidad única
- Principio Abierto/Cerrado (OCP)
- Principio de Sustitución de Liskov (LSP)
- Principio de Segregación de Interfaces (ISP)



}

NOTA {

DIP es fundamental para la implementación de los otros principios SOLID. Ayuda a mantener el código abierto para la extensión (OCP), garantiza que las implementaciones puedan ser sustituidas (LSP), promueve interfaces específicas (ISP) y mantiene las responsabilidades claras (SRP).

}

Técnicas de Inyección de Dependencias - Inyección de Constructor{

```
from abc import ABC, abstractmethod

class Notificador(ABC):
    @abstractmethod
    def enviar(self, mensaje: str):
        pass
class EmailNotificador(Notificador):
    def enviar(self, mensaje: str):
        print(f'Enviando email: {mensaje}')
class App:
    def __init__(self, notificador: Notificador):
        self.notificador = notificador

    def enviar_notificacion(self, mensaje: str):
        self.notificador.enviar(mensaje)
        print("Notificación enviada correctamente.")
notificador = EmailNotificador()
app = App(notificador)
app.enviar_notificacion("Hola Mundo!")
```

{}

Técnicas de Inyección de Dependencias - Inyección de Setter (Método){

```
class App:
    def __init__(self):
        self.notificador = None

    def set_notificador(self, notificador: Notificador):
        self.notificador = notificador

    def enviar_notificacion(self, mensaje: str):
        if self.notificador:
            self.notificador.enviar(mensaje)
            print("Notificación enviada correctamente.")
        else:
            print("Notificador no configurado.")

app = App()
notificador = EmailNotificador()
app.set_notificador(notificador)
app.enviar_notificacion("Hola Mundo!")
```

{}

Técnicas de Inyección de Dependencias - Inyección de Método {

```
class App:  
    def enviar_notificacion(self, notificador: Notificador, mensaje: str):  
        notificador.enviar(mensaje)  
        print("Notificación enviada correctamente.")  
  
app = App()  
notificador = EmailNotificador()  
app.enviar_notificacion(notificador, "Hola Mundo!")
```

}

<!--SOLID-->

Codifiquemos! {

<Ejemplo/>

}

```
<!--SOLID-->
```

Gracias {

```
<Por="Raquel Martinez/>
```

```
}
```



```
<!--SOLID-->
```

Herencia Vs Composición

{

```
<Por="Raquel Martinez"/>
```

```
}
```

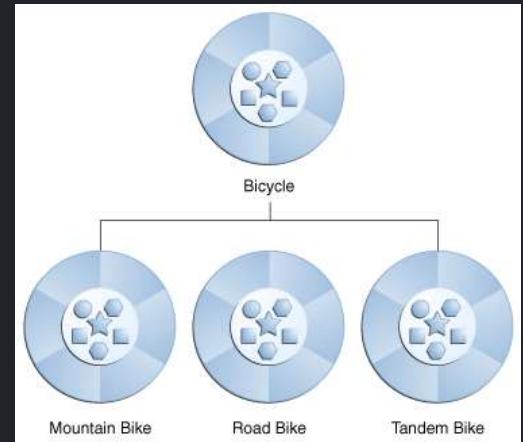


Contenidos

- 01 Herencia Vs Composición
- 02 Principios de diseño
- 03 Cohesión
- 04 Acoplamiento

Herencia {

es un / es una



}

Ventajas de la herencia {

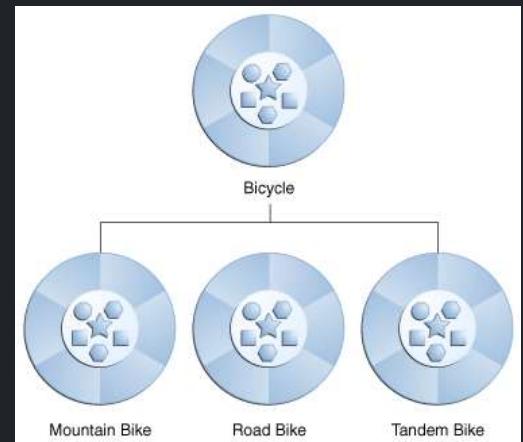
- Reutilización de código
- Polimorfismo
- Sobreescritura de métodos
- Desarrollo guiado

```
class Figura:  
    def calcular_area(self):  
        return 0  
  
class Rectangulo(Figura):  
    def calcular_area(self):  
        return self.ancho * self.altura
```

}

Desventajas de la herencia {

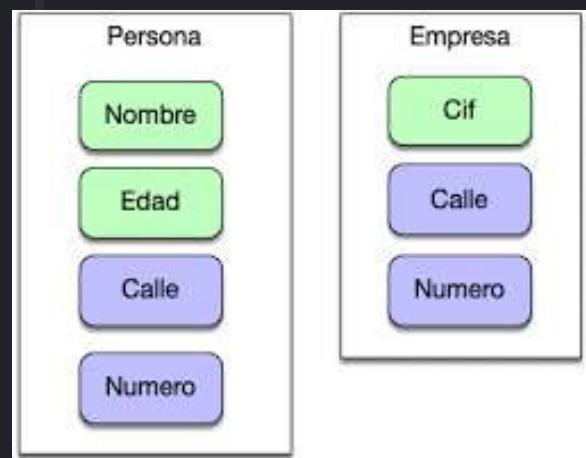
- Proyectos ágiles
- "la fiesta de los override"
- "la herencia de 7 niveles"



}

Composición {

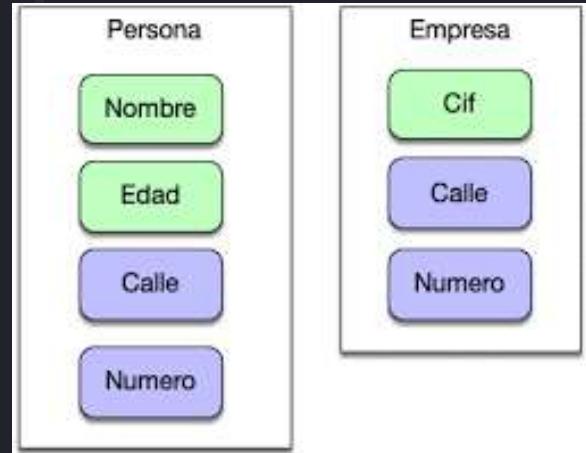
- Composición quiere decir que tenemos una instancia de una clase que contiene instancias de otras clases que implementan las funciones deseadas.



}

Ventajas de la composición {

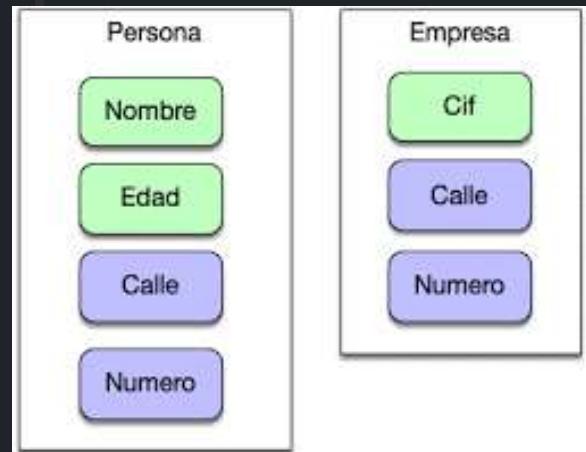
- Versatilidad
- Creación/destrucción dinámica
- Polimorfismo mediante interfaces



}

Composición cuando? {

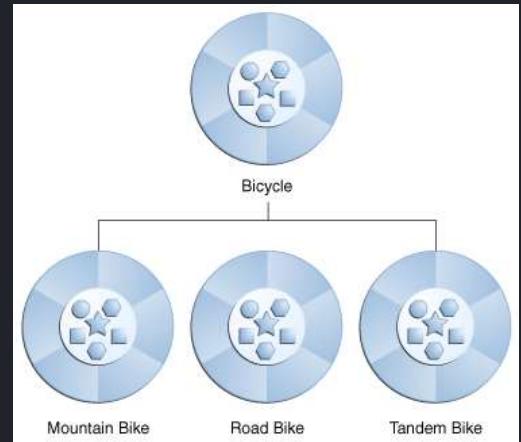
- No existe una relación de tipo "es un" clara y permanente
- Se necesita flexibilidad y capacidad de cambio
- Se busca desacoplar y reutilizar componentes
- Se requiere implementar comportamientos intercambiables
- Se tiene una relación de tipo "tiene un"



}

Herencia cuando? {

- Existe una relación de tipo "es un" clara y permanente
- Se busca reutilizar código y comportamiento común
- Se necesita polimorfismo estático
- Se modela el dominio del problema



}

<!--SOLID-->

Codifiquemos! {

<Ejemplo/>

}

Principios de diseño {

Que son?

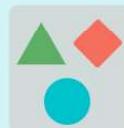
Cohesión y el acoplamiento

Los principios de diseño en software son directrices generales que ayudan a los desarrolladores a crear sistemas de software que sean fáciles de entender, mantener y extender.

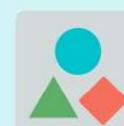
}

Cohesión {

- La cohesión se refiere a la medida en que las responsabilidades y funciones de un módulo (clase, método o componente) están relacionadas y enfocadas en una única tarea o propósito.



Baja cohesión

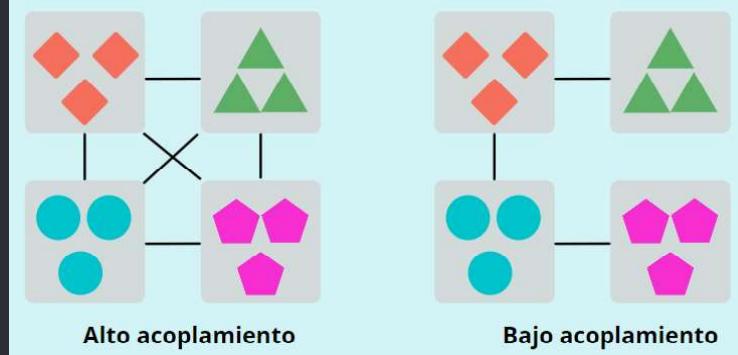


Alta cohesión

}

Acoplamiento {

- Grado de interdependencia entre los diferentes módulos de un sistema.



}

Alta cohesión

Facilita la comprensión del código

Promueve la reutilización de código

Simplifica las pruebas y el mantenimiento

}

Bajo acoplamiento

Permite que los módulos sean más independientes

Facilita el mantenimiento y la evolución del código

Promueve la modularidad y la separación de preocupaciones

}

<!--SOLID-->

Gracias {

<Por="Raquel Martinez/>

}