Universidad Intercontinental de la Empresa

Asignatura

# Arquitectura del Software 

Programación Orientada a Objetos

Profesor

**Yago Fontenla Seco**

{*yago.fontenla1@uie.edu*}

# Programación Orientada a Objetos

> Paradigma en el que un programa se estructura en torno a **objetos y clases**. Los objetos son **instancias** de clases que contienen datos (**atributos**) y comportamiento (**métodos**).

Python permite implementar POO mediante la definición de clases, la creación de objetos y la aplicación de conceptos clave como método, herencia, encapsulamiento, y polimorfismo.

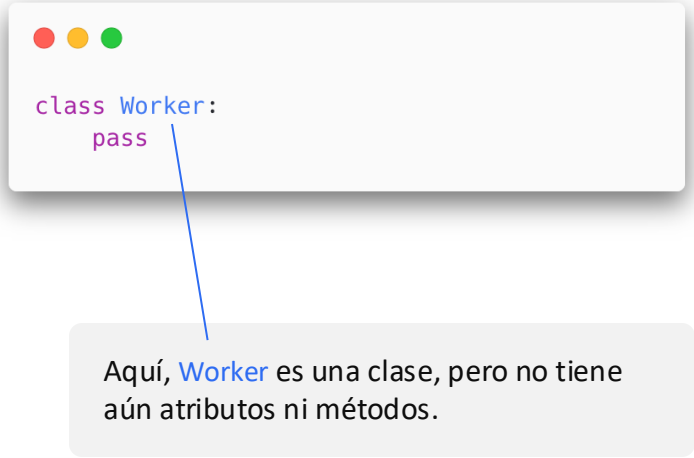
**Conceptos clave:**
- Clase
- Objeto
- Atributo
- Método
- Herencia
- Encapsulamiento
- Polimorfismo

# Clase

> Una clase es un molde o plantilla a partir de la cual se crean objetos.

Define las propiedades (atributos) y el comportamiento (métodos) que compartirán todas las instancias de esa clase.

En términos simples, es una descripción general de un tipo de objeto.



# Objeto

Los objetos son instancias de una clase. Cada objeto tiene sus propios valores para los atributos definidos en la clase.

Representa una entidad individual que tiene su propio estado (definido por los atributos) y comportamiento (definido por los métodos).

Cada objeto creado a partir de la clase tiene los mismos atributos y métodos definidos en la clase, pero sus valores y comportamientos pueden ser únicos.

```python
class Worker:
    pass

fry = Worker()
```

Con esta línea de código creamos el objeto fry, de clase Worker.

# Atributos

Los **atributos** son las propiedades o características de un objeto. En POO, los atributos pueden ser de **clase** o de **instancia.**

- **Atributos de instancia**: Se definen dentro del método __init__(). Cada objeto puede tener diferentes valores para sus atributos de instancia, lo que permite que diferentes objetos se comporten de manera distinta.

- **Atributos de clase**: Son compartidos por todas las instancias de la clase. Se definen a nivel de clase, fuera de cualquier método. Un cambio en un atributo de clase afectará a todas las instancias de esa clase.

```python
class Worker:
    workplace = 'Planet Express'  # Atributo de clase

    def __init__(self, name, age):
        self.name = name  # Atributo de instancia
        self.age = age    # Atributo de instancia

# Instanciamos la clase
fry = Worker(name="Fry", age=35)
profe = Worker(name="Yago", age=28)
```

# Self

El concepto de self en Python es un elemento fundamental en la Programación Orientada a Objetos (POO). self se refiere a la **instancia actual de la clase** y es necesario para acceder a los **atributos y métodos** de esa instancia desde dentro de la propia clase.

**¿Qué es self?**

self es el primer parámetro que deben tener los métodos de instancia en Python, aunque no es necesario que el programador lo pase explícitamente cuando llama al método. Python lo hace automáticamente. Este parámetro representa al **objeto específico** sobre el cual se está llamando un método o accediendo a un atributo.

**El parámetro self le permite a Python saber a qué objeto en particular se está refiriendo y trabajar con los datos asociados a esa instancia.**

# Self

El concepto de self en Python es un elemento fundamental en la Programación Orientada a Objetos (POO). self se refiere a la **instancia actual de la clase** y es necesario para acceder a los **atributos y métodos** de esa instancia desde dentro de la propia clase.

```python
class Worker:
    workplace = 'Planet Express'  # Atributo de clase

    def __init__(self, name, age):
        self.name = name  # Atributo de instancia
        self.age = age     # Atributo de instancia

# Instanciamos la clase
fry = Worker(name="Fry", age=35)
profe = Worker(name="Yago", age=28)
```

# Modificación de atributos

Los **atributos** son las propiedades o características de un objeto. En POO, los atributos pueden ser de **clase** o de **instancia.**

- **Atributos de instancia**: Si cambias el valor de un atributo de instancia, solo afectará al objeto individual.
- **Atributos de clase**: Son compartidos por todas las instancias de la clase. Se definen a nivel de clase, fuera de cualquier método. Un cambio en un atributo de clase afectará a todas las instancias de esa clase.

```
profe.age = 29
print(yago.age)  # Output: 29 (solo afecta a profe)
print(fry.age)  # Output: 35 (no afecta a car2)

Worker.workplace = "UIE"
print(yago.workplace)  # Output: "UIE"
print(fry.workplace)  # Output: "UIE"
```

# Métodos

> Los métodos son funciones que se definen dentro de una clase y que pueden ser llamadas por los objetos de esa clase.

Los métodos permiten definir el comportamiento que los objetos de una clase pueden tener, y pueden interactuar con los atributos del objeto, modificarlos, o devolver valores en función de esos atributos.

- **Métodos especiales**
- **Métodos de clase**
- **Métodos estáticos**
- **Métodos de instancia**

```python
class Worker:
    workplace = 'Planet Express'

    def __init__(self, name, age):
        self.name = name
        self.age = age

    # Metodo
    def say_hi(self):
        return f"Hi, I'm {self.name}"
```

# Métodos especiales

> Los métodos especiales, “métodos mágicos” o “dunder methods” en Python comienzan y terminan con “__”

Uno de los más importantes es __init__(), que se usa para inicializar los objetos.

__init__()
__str__()
__eq__()

Estos métodos permiten definir y personalizar el comportamiento de los objetos cuando se utilizan operadores, se imprimen o se convierten en cadenas de texto, y otras operaciones integradas de Python.

```python
class Worker:

    # Atributo de clase
    workplace = 'Planet Express'

    def __init__(self, name, age, identifier):
        # Atributos de instancia
        self.name = name
        self.age = age
        self.identifier = identifier

    # Metodo de instancia
    def say_hi(self, greetings):
        return f"Hi, I'm {self.name} and I say {greetings}"

    # Metodo especial
    def __str__(self):
        return f"{self.name} is {self.age} years old."

    # Metodo especial de comparacion
    def __eq__(self, other):
        if isinstance(other, Worker):
            return self.identifier == other.identifier
        return False
```

# Herencia

La **herencia** es uno de los conceptos clave en la Programación Orientada a Objetos (POO). Permite que una clase llamada **clase hija** o **subclase** herede atributos y métodos de otra clase llamada **clase padre** o **superclase**.

Esto fomenta la reutilización del código y facilita la extensión y modificación del comportamiento sin necesidad de reescribir todo desde cero.

Cuando una clase hereda de otra, la clase hija tiene acceso a:

- Todos los **atributos** de la clase padre.
- Todos los **métodos** de la clase padre.

```
class Human(Worker):
    pass

class Decapodian(Worker):
    def say_hi(self, greetings="w00p w00p w00p"):
        return f"Hi, I'm {self.name} and I say {greetings}"
```

Con pass, hacemos que Human sea idéntico a la clase padre Worker sin implementar ninguna diferencia.

# Polimorfismo

> 💡 El **polimorfismo** es otro principio clave de la POO que permite que los objetos de diferentes clases respondan a los mismos métodos, pero de manera diferente.

El polimorfismo permite que el **mismo nombre de método** o función **se comporte de forma distinta** según el tipo de objeto que lo esté utilizando.

**Tipos de polimorfismo:**

- **Polimorfismo basado en clases** (sobreescritura): Las clases hijas pueden sobrescribir los métodos de la clase padre para proporcionar una implementación diferente.
- **Polimorfismo basado en funciones** (polimorfismo de funciones): En Python, una función puede trabajar con diferentes tipos de datos y producir resultados distintos según el tipo de entrada.

# Sobreescritura

> La **sobreescritura de métodos** es el proceso mediante el cual una clase hija redefine o modifica un método heredado de su clase padre.

Esto es útil cuando necesitas que la clase hija tenga un comportamiento diferente para un método específico.

En este caso, cuando se llama al método desde una instancia de la clase hija, **se ejecuta la versión del método definida en la clase hija**, no la de la clase padre.

```python
class Human(Worker):
    pass

class Decapodian(Worker):
    def say_hi(self, greetings="w00p w00p w00p"):
        return f"Hi, I'm {self.name} and I say {greetings}"

class Robot(Worker):
    def say_hi(self, greetings="010101"):
        return f"Hi, I'm {self.name} and I say {greetings}"
```

En la clase Robot, hija de Worker, volvemos a definir el método say_hi() para que tenga un comportamiento diferente al de la clase padre.

# Sobreescritura – Super()

La **sobreescritura de métodos** es el proceso mediante el cual una clase hija redefine o modifica un método heredado de su clase padre.

El uso de super() es útil cuando necesitas que la clase hija tenga un comportamiento diferente para un método específico, **pero aún deseas aprovechar la estructura general de la clase padre**.

La clase hija Human extiende el constructor de la clase padre Worker lo que permite que se inicialicen los atributos name y age definidos en la clase padre. Además, se añade el nuevo atributo department que es específico de la clase Human.

```python
class Worker:
    workplace = 'Planet Express'

    def __init__(self, name, age):
        self.name = name
        self.age = age

    def say_hi(self):
        return f"Hi, I'm {self.name}"

# Clase hija con un nuevo atributo "department"
class Human(Worker):
    def __init__(self, name, age, department):
        super().__init__(name, age) # Llamada al constructor de la clase padre
        self.department = department # Atributo adicional en la clase hija

    def say_hi(self):
        # Extiende la funcionalidad del saludo para incluir el departamento
        return f"Hi, I'm {self.name} from {self.department} department."
```

# Polimorfismo en funciones

En Python, una función **puede trabajar con diferentes tipos de datos** y producir resultados distintos según el tipo de entrada.

En este ejemplo, la función area() es polimórfica porque opera sobre diferentes tipos de objetos (Circle y Rectangle) y ajusta su comportamiento en función del tipo de objeto recibido.

```python
class Shape:
    def area(self):
        raise NotImplementedError("Subclasses must implement this method")


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

    def area(self):
        return 3.14 * self.radius ** 2


class Rectangle(Shape):
    def __init__(self, width, height):
        self.width = width
        self.height = height

    def area(self):
        return self.width * self.height


# Creación de objetos
my_circle = Circle(5)
my_rectangle = Rectangle(3, 4)

# La misma llamada funciona para ambos objetos
print(my_circle.area())     # Output: 78.5
print(my_rectangle.area())  # Output: 12
  return go(f, seed, [])
}
```

# Encapsulamiento

> El **encapsulamiento** restringe el acceso a los atributos de un objeto desde fuera de su clase, permitiendo que sólo se pueda acceder a ellos a través de métodos públicos específicos (getters y setters).

El encapsulamiento ayuda a **proteger** los atributos de un objeto y **controlar** cómo se modifican.

En Python, no existen atributos estrictamente privados, pero se utiliza la convención de prefijar un guion bajo (_) o doble guion bajo (__) a los nombres de atributos o métodos para indicar que son privados y no deberían ser accedidos directamente desde fuera de la clase.

**Tipos de encapsulamiento:**

- **Atributos "protegidos"** (un guion bajo _): Indica que el atributo no debería ser accedido directamente, aunque no es estrictamente privado.
- **Atributos "privados"** (doble guion bajo __): Python renombra el atributo para que no pueda ser accedido fácilmente desde fuera de la clase (name mangling).

# Encapsulamiento

> El **encapsulamiento** restringe el acceso a los atributos de un objeto desde fuera de su clase, permitiendo que sólo se pueda acceder a ellos a través de métodos públicos específicos (getters y setters).

En este ejemplo, el atributo `__ssn` es privado y sólo puede ser accedido a través del método `get_ssn()`. Esto asegura que el número de seguridad social no se modifique directamente desde fuera de la clase. El atributo `_age` es protegido y debería ser accedido o modificado mediante un método setter como `set_age()`.

```python
class Person:
    def __init__(self, name, age):
        self.name = name           # Atributo público
        self._age = age            # Atributo protegido (convención)
        self.__ssn = "123-45-6789" # Atributo privado

    # Método getter para el atributo privado
    def get_ssn(self):
        return self.__ssn

    # Método setter para cambiar el valor de un atributo
    def set_age(self, new_age):
        if new_age > 0:
            self._age = new_age
        else:
            print("La edad no puede ser negativa.")

# Creación de un objeto
person = Person("John", 30)

# Acceso a un atributo público
print(person.name)  # Output: John

# Acceso a un atributo protegido (aunque es posible acceder, no debería hacerse
directamente)
print(person._age)  # Output: 30

# Acceso a un atributo privado (genera error si se intenta acceder directamente)
# print(person.__ssn)  # Esto genera un error

# Acceso al atributo privado a través del getter
print(person.get_ssn())  # Output: 123-45-6789

# Modificación del atributo protegido a través del setter
person.set_age(35)
print(person._age)  # Output: 35
```

Universidad Intercontinental de la Empresa

Asignatura

# Arquitectura del Software

Profesor

**Yago Fontenla Seco**

{*yago.fontenla1@uie.edu*}