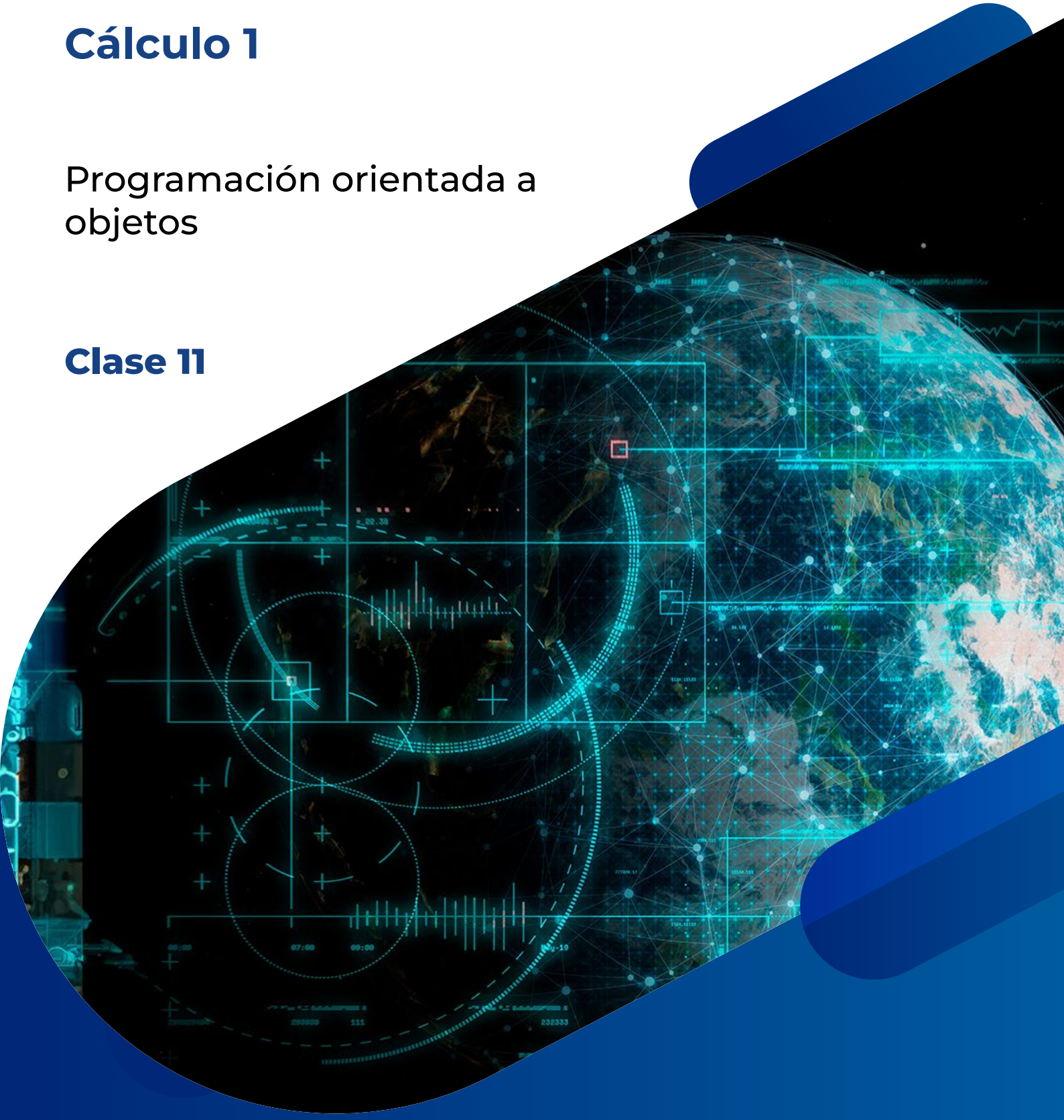
Cálculo 1

Programación orientada a objetos

Clase 11



Ingeniería en ciberseguridad

La excelencia no se improvisa

Clase 11: Programación orientada a objetos

La **Programación Orientada a Objetos** (POO) es un paradigma de programación que organiza el diseño del software alrededor de datos u "objetos", en lugar de funciones y lógica. Este enfoque facilita la creación de aplicaciones y sistemas más modularizados, reutilizables y manejables. En Python, la POO es ampliamente utilizada debido a su flexibilidad y simplicidad.

11.1 Conceptos fundamentales de la POO

La POO se basa en cuatro conceptos fundamentales: clases, objetos, herencia y polimorfismo.

Clases y objetos:

Una **clase** es una plantilla o modelo que define un conjunto de atributos y métodos que caracterizan a cualquier objeto de ese tipo. Los atributos representan los datos, mientras que los métodos representan el comportamiento.

Un **objeto** es una instancia de una clase. Cada objeto puede tener diferentes valores para los atributos definidos en su clase.

Ejemplo:

Figura 35

Ejemplo de clase en Python

```python
class Coche:
    def __init__(self, marca, modelo):
        self.marca = marca
        self.modelo = modelo

    def describir(self):
        return f"Coche: {self.marca} {self.modelo}"

mi_coche = Coche("Toyota", "Corolla")
print(mi_coche.describir())
```

Nota. Ejemplo de una clase (coche)

Herencia: La herencia permite que una clase (subclase) herede atributos y métodos de otra clase (superclase), lo que facilita la reutilización del código y la creación de una jerarquía de clases.

Ejemplo:

Figura 36

Ejemplo de Herencia

```python
class Vehiculo:
    def __init__(self, marca):
        self.marca = marca

class Coche(Vehiculo):
    def __init__(self, marca, modelo):
        super().__init__(marca)
        self.modelo = modelo
```

Nota. En este ejemplo se muestra la creación de una superclase denominada vehículo.

Polimorfismo: El polimorfismo permite que diferentes clases utilicen la misma interfaz para métodos que pueden comportarse de manera diferente, lo que es fundamental para la flexibilidad y escalabilidad del código.

Figura 37

Ejemplo de poliformismo

```python
class Animal:
    def hacer_sonido(self):
        raise NotImplementedError

class Perro(Animal):
    def hacer_sonido(self):
        return "Ladrar"

class Gato(Animal):
    def hacer_sonido(self):
        return "Maullar"

def sonido_animal(animal):
    print(animal.hacer_sonido())

sonido_animal(Perro())  # Output: Ladrar
sonido_animal(Gato())   # Output: Maullar
```

Nota. Se muestra un ejemplo de uso de diferentes interfaces para métodos

1.2 Ventajas de la POO

- **Modularidad:** El código se organiza en módulos, lo que facilita su gestión y mantenimiento.
- **Reutilización:** A través de la herencia, se puede reutilizar código existente en nuevas aplicaciones.
- **Manejabilidad:** La POO facilita el trabajo en equipo, ya que el código es más comprensible y se pueden asignar módulos específicos a diferentes desarrolladores.
- **Escalabilidad:** Es más fácil ampliar sistemas existentes con nuevas características.

Python es un lenguaje altamente compatible con la POO. Su sintaxis simple y sus potentes bibliotecas lo hacen ideal para desarrollar aplicaciones de software que requieren una arquitectura basada en objetos, como juegos, aplicaciones de escritorio y sistemas complejos de procesamiento de datos (López & Torres, 2019).

11.3 Funciones en Python

En Python, las funciones son bloques de código reutilizables diseñados para realizar una tarea específica. Las funciones permiten la modularidad del código y mejoran la legibilidad y mantenibilidad de los programas. A continuación, se describen los conceptos fundamentales sobre las funciones en Python y se proporcionan ejemplos prácticos.

Una función en Python se define usando la palabra clave def, seguida del nombre de la función, paréntesis y un bloque de código indentado. Las funciones pueden aceptar argumentos y devolver valores.

Figura 38

Sintaxis básica

```
def nombre_de_la_función(argumentos):
    # bloque de código
    return valor
```

Nota. En la gráfica se muestra la sintaxis básica de funciones

Las funciones pueden aceptar varios tipos de argumentos:

- **Argumentos posicionales:** Los valores se asignan a los parámetros en el orden en que se pasan.
- **Argumentos nombrados:** Los valores se asignan a los parámetros utilizando sus nombres.
- **Argumentos por defecto:** Los parámetros pueden tener valores por defecto que se utilizan si no se pasa un valor para ese parámetro.
- **Argumentos variables:** Se pueden usar para aceptar un número variable de argumentos (*args y **kwargs).

Figura 39

Ejemplo de argumentos

```
def mostrar_info(nombre, edad=30, *args, **kwargs):
    print(f"Nombre: {nombre}")
    print(f"Edad: {edad}")
    if args:
        print(f"Otros argumentos: {args}")
    if kwargs:
        print(f"Otros argumentos nombrados: {kwargs}")

mostrar_info("Luis", 25, "Ingeniero", ciudad="Madrid", país="España")
```

Nota. Código para declarar argumentos en Python

Las **funciones lambda**, también conocidas como funciones anónimas, son funciones pequeñas y sin nombre definidas con la palabra clave lambda. Son útiles para realizar operaciones rápidas y simples.

Figura 40

Función lambda

```
lambda argumentos: expresión
```

Nota. Declaración de la función lambda

Las **funciones de orden superior** son aquellas que aceptan otras funciones como argumentos o devuelven una función como resultado. Son fundamentales en la programación funcional.

Figura 41

Funciones de orden superior

```python
def aplicar_funcion(func, valor):
    return func(valor)

print(aplicar_funcion(doblar, 10))  # Usando la función lambda 'doblar'
```

Nota. Ejemplo para aplicar funciones de orden superior

Decoradores

Los decoradores son una forma de modificar el comportamiento de una función o método. Se definen como funciones que envuelven a otra función para extender su comportamiento.

Figura 42

Decoradores

```python
def decorador_saludo(func):
    def envoltura(nombre):
        print("Hola!")
        func(nombre)
        print("Adiós!")
    return envoltura

@decorador_saludo
def saludar(nombre):
    print(f"Hola, {nombre}")

saludar("Ana")
```

Nota. Ejemplo definición de decoradores en Python

En el análisis de datos, las funciones se utilizan para limpiar, procesar y analizar grandes conjuntos de datos. Bibliotecas como pandas y numpy dependen en gran medida de las funciones para operar sobre estructuras de datos.

En el desarrollo web, las funciones se utilizan para manejar solicitudes y respuestas HTTP, procesar datos de formularios y generar contenido dinámico.

11.3 Clases en Python y ejercicios de aplicación

En Python, las clases son una de las características más importantes de la Programación Orientada a Objetos (POO). Las clases permiten crear nuevos tipos de objetos que agrupan datos y funcionalidades.

A continuación, se describen los conceptos fundamentales de las clases en Python, sus componentes y su uso en aplicaciones prácticas.

Una clase es una plantilla para crear objetos (instancias de clases). Los objetos son instancias concretas de las clases y representan entidades con propiedades y comportamientos definidos por la clase.

Figura 43

Ejemplo de Clase

```python
class NombreDeLaClase:
    def __init__(self, parametros):
        self.atributo1 = valor1
        self.atributo2 = valor2

    def metodo(self):
        # código del método
        pass
```

Nota. Ejemplo para practica de clases en Python

Componentes de una clase

Método constructor

El método __init__ es el constructor de la clase y se llama automáticamente cuando se crea una nueva instancia. Inicializa los atributos del objeto.

Figura 44

*Método constructor _init_*

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

Nota. Ejemplo de definición de clase _init_

Atributos de Instancia:

Los atributos de instancia son variables que pertenecen a cada instancia de una clase. Se definen dentro del método constructor __init__.

Figura 45

Atributos de instancia

```
class Animal:
    def __init__(self, especie):
        self.especie = especie
```

Nota. En el ejemplo anterior se muestra a definición de la clase Animal

Métodos:

Los métodos son funciones definidas dentro de una clase que operan sobre los atributos de la instancia. Se definen como cualquier otra función, pero su primer parámetro es siempre self, que representa la instancia.

Figura 46

Primer parámetro Self

```
class Circulo:
    def __init__(self, radio):
        self.radio = radio

    def area(self):
        return 3.14 * self.radio ** 2
```

Nota. Definición del primer parámetro Self

Atributos y métodos de clase:

Los atributos de clase son variables que pertenecen a la clase y son compartidos por todas las instancias. Se definen fuera de cualquier método.

Figura 47

Atributos en clases

```python
class Mascota:
    numero_de_patas = 4  # Atributo de clase

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

Nota. Ejemplo de atributo de clase mascota “4”

Herencia

La herencia permite crear una nueva clase basada en una clase existente. La nueva clase (subclase) hereda atributos y métodos de la clase base (superclase).

Figura 48

Ejemplo de herencia en clases

```python
class Vehiculo:
    def __init__(self, marca, modelo):
        self.marca = marca
        self.modelo = modelo

class Coche(Vehiculo):
    def __init__(self, marca, modelo, puertas):
        super().__init__(marca, modelo)
        self.puertas = puertas
```

Nota. Gráfica de dos clases que mantienen una herencia en común

Encapsulamiento

El encapsulamiento es el principio de ocultar los detalles internos de la implementación de una clase. En Python, los atributos y métodos privados se denotan con un guion bajo _.

Figura 49

Encapsulamiento

```python
class CuentaBancaria:
    def __init__(self, titular, saldo):
        self._titular = titular  # Atributo privado
        self._saldo = saldo      # Atributo privado

    def depositar(self, monto):
        self._saldo += monto

    def retirar(self, monto):
        if monto <= self._saldo:
            self._saldo -= monto
        else:
            print("Saldo insuficiente")

    def obtener_saldo(self):
        return self._saldo
```

Nota. Ejemplo de encapsulamiento en Python

Polimorfismo

El polimorfismo permite que diferentes clases utilicen la misma interfaz. En Python, esto se puede lograr mediante la herencia y la sobreescritura de métodos.

Figura 50

Ejemplo de Interfaz compartida

```python
class Animal:
    def hacer_sonido(self):
        pass

class Perro(Animal):
    def hacer_sonido(self):
        return "Ladrar"

class Gato(Animal):
    def hacer_sonido(self):
        return "Maullar"

def sonido_animal(animal):
    print(animal.hacer_sonido())

sonido_animal(Perro())  # Output: Ladrar
sonido_animal(Gato())   # Output: Maullar
```

Nota. Ejemplo interfaz clase animal

Referencias

Gómez, M., & Rodríguez, J. (2019). *GeoGebra y su aplicación en la enseñanza de las matemáticas*. Editorial Educativa.

Martínez, S., & Torres, A. (2020). *Simulación y modelado en ingeniería: Herramientas y aplicaciones*. Editorial Técnica.

Pérez, L., & López, J. (2021). *Tecnologías educativas: Integración de software en la enseñanza de las ciencias*. Editorial Académica.

Glosario de los términos citados

Encapsulamiento: Es un principio de la programación orientada a objetos que restringe el acceso directo a algunos de los componentes de un objeto. Los atributos y métodos privados se ocultan y solo se pueden acceder a través de métodos públicos definidos en la clase. Esto mejora la modularidad y protege la integridad de los datos.

Interfaz Compartida: Es un conjunto de métodos que diferentes clases pueden implementar, lo que permite que los objetos de estas clases sean tratados de manera uniforme. En la programación orientada a objetos, una interfaz define un contrato que las clases deben cumplir, asegurando que los métodos especificados en la interfaz estén presentes en las clases que la implementan.

PUCE Virtual

La excelencia no se improvisa

síguenos

ICYS - 12 - 241