Temario:

1. Introducción a la programación orientada a objetos en Python
    - ¿Qué es la POO?
    - ¿Qué es la abstracción en POO?
2. Fundamentos de la POO
    - Clases y objetos en Python
        i. Definición de clases
        ii. Creación de objetos
        iii. Atributos y métodos
        iv. Constructores y destructores
        v. Modificadores de acceso
    - Encapsulamiento en Python
        i. Definición de encapsulamiento
        ii. Acceso a los atributos y métodos privados
        iii. Getter y Setter
    - Herencia en Python
        i. Definición de herencia
        ii. Clases padre y clases hijas
        iii. Métodos y atributos heredados
        iv. Sobrecarga de métodos
    - Polimorfismo en Python
        i. Definición de polimorfismo
        ii. Tipos de polimorfismo
        iii. Sobrecarga y sobreescritura de métodos
3. Relaciones entre clases
    - Asociación
    - Agregación
    - Composición
4. Implementación de POO en Python. Ejercicios.
    - Uso de clases y objetos en Python
    - Herencia y polimorfismo en Python
    - Encapsulamiento en Python
5. Excepciones
    - Definición de excepciones
    - Manejo de excepciones
    - Lanzamiento de excepciones
6. Implementación de POO en Python. Ejercicios.
    - Practica con excepciones
7. Diseño de clases en Python
    - Cohesión y acoplamiento en Python
    - Principios SOLID en Python
        i. S - Principio de Responsabilidad Única
        ii. O - Principio de Abierto/Cerrado
        iii. L - Principio de Sustitución de Liskov
        iv. I - Principio de Segregación de Interfaces
        v. D - Principio de Inversión de Dependencias
8. Programación orientada a objetos avanzada: Clases abstractas
    - Interfaces

        i. Definición de clases abstractas
        ii. Métodos abstractos
9. Aplicaciones prácticas de POO en Python. Ejercicios
    - Desarrollo de aplicaciones con POO en Python
10. Proyecto final: Aplicación de una API REST con flask
    - Flask y API REST
        i. Introducción a Flask
        ii. Creación de rutas y vistas en Flask
        iii. Creación de API RESTful en Flask
        iv. Pruebas con Postman
    - Desarrollo de un proyecto completo aplicando los conceptos de POO aprendidos en Python
    - Uso del framework Flask para el desarrollo del proyecto en Python

Introducción a la programación orientada a objetos en Python

¿Qué es la POO?

Es un paradigma de programación que proporciona un medio para estructurar un programa, en la cual, las propiedades y comportamientos se agrupen en individuos, o sea, en objetos.

La programación orientada a objetos surge en la historia como un intento para dominar la complejidad que, de forma innata, posee el software. Tradicionalmente, la forma de enfrentarse a esta complejidad ha sido empleando lo que llamamos programación estructurada, que consiste en descomponer el problema objeto de resolución en subproblemas y más subproblemas hasta llegar a acciones muy simples y fáciles de codificar. Se trata de descomponer el problema en acciones, en verbos. En el ejemplo de un programa que resuelva ecuaciones de segundo grado, descomponíamos el problema en las siguientes acciones: primero, pedir el valor de los coeficientes a, b y c; después, calcular el valor del discriminante; y por último, en función del signo del discriminante, calcular ninguna, una o dos raíces.

Como podemos ver, descomponíamos el problema en acciones, en verbos; por ejemplo el verbo pedir, el verbo hallar, el verbo comprobar, el verbo calcular…

La programación orientada a objetos es otra forma de descomponer problemas. Este nuevo método de descomposición es la descomposición en objetos; vamos a fijarnos no en lo que hay que hacer en el problema, sino en cuál es el escenario real del mismo, y vamos a intentar simular ese escenario en nuestro programa.

Dicho de otra manera, la programación orientada a objetos es un enfoque para modelar cosas concretas del mundo real, como automóviles, así como relaciones entre cosas, como empresas y empleados, estudiantes y maestros, y así. OOP modela entidades del mundo real como objetos de software que tienen algunos datos asociados con ellos y pueden realizar ciertas funciones.

¿Qué es la abstracción en POO?

Es uno de los pilares fundamentales de la programación orientada a objetos (POO). En términos simples, la abstracción es el proceso de enfocarse en las características esenciales de un objeto o concepto, ignorando los detalles no importantes.

Fundamentos de la POO

Clases y objetos en Python

Clases

Las clases proporcionan un medio para agrupar datos y funcionalidades. Una **clase** es una plantilla. Define de manera genérica cómo van a ser los objetos de un determinado tipo.

El uso de clases permite tener múltiples instancias de la misma clase. Estas se comportan de la forma que se ha definido en la clase,

Objeto

Un objeto es una entidad que se crea tomando como base una clase de programación, el proceso de creación ocurre durante la ejecución del programa y se lo conoce como instanciación. El objeto adquiere un estado durante su creación y puede ir modificándose durante todo el tiempo de ejecución hasta que el programa finalice o se pierda la referencia de ese objeto y sea eliminado por el recolector de basura.

Atributos

Las instancias o las clases pueden guardar información lo que se denomina atributos.. los atributos son variables presentes en objetos o en clases, y se encargan de guardar información. Pueden guardar no solo valores, sino también funciones asignadas tras la creación de la instancia u otros objetos. Hay dos tipos de atributos, atributos de clases y atributos de instancia.

- *Atributos de instancia*: son atributos que pertenecen a una instancia específica de una clase. Cada objeto tiene su propio conjunto de valores para los atributos de

instancia. Estos atributos se definen dentro del método constructor de la clase utilizando la sintaxis self.nombre_atributo.

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

mi_coche = Coche("Ford", "Mustang", 2022)
print(mi_coche.marca)  # Output: Ford
print(mi_coche.modelo)  # Output: Mustang
print(mi_coche.anio)  # Output: 2022
```

- *Atributos de clase*: son atributos que pertenecen a la clase en sí, en lugar de pertenecer a una instancia específica de la clase. Estos atributos se definen fuera de los métodos de la clase, pero dentro de la clase misma

```python
class Coche:
    num_ruedas = 4  # Atributo de clase

    def __init__(self, marca, modelo, anio):
        self.marca = marca  # Atributo de instancia
        self.modelo = modelo  # Atributo de instancia
        self.anio = anio  # Atributo de instancia

mi_coche = Coche("Ford", "Mustang", 2022)
print(mi_coche.marca)  # Output: Ford

otro_coche = Coche("Chevrolet", "Camaro", 2022)
print(otro_coche.num_ruedas)  # Output: 4
```

Métodos

En Python, un método es una función definida dentro de una clase y que se utiliza para realizar operaciones en los objetos de esa clase. Los métodos pueden acceder y modificar los atributos de un objeto, y pueden tomar argumentos como cualquier otra función en Python.

Los métodos en Python tienen un primer argumento especial llamado self, que hace referencia al objeto en sí mismo. Este argumento se usa para acceder a los atributos de la instancia y llamar a otros métodos en la misma instancia.

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

    def saludar(self):
        print(f"Hola, mi nombre es {self.nombre} y tengo {self.edad} años.")

persona1 = Persona("Juan", 25)
persona1.saludar()  # Output: Hola, mi nombre es Juan y tengo 25 años.
```

Constructor

Un constructor es un método especial que se llama cuando se crea un objeto de una clase. El constructor se usa para inicializar los atributos de la instancia y puede tener cualquier nombre, pero comúnmente se llama __init__.

```python
class Persona:
    def __init__(self, nombre, edad):#Contructor de la clase Persona
        self.nombre = nombre
        self.edad = edad

persona1 = Persona("Juan", 25)
print(persona1.nombre)  # Output: Juan
print(persona1.edad)  # Output: 25
```

Encapsulamiento en Python

Se refiere a la capacidad de una clase para ocultar y proteger su implementación interna, permitiendo que sólo su interfaz pública sea accesible a otros objetos. Esto se logra mediante la definición de atributos y métodos privados en una clase, que solo pueden ser accedidos y modificados por métodos públicos específicos de la clase. De esta manera, la implementación interna de la clase se mantiene oculta y protegida, lo que facilita el mantenimiento y la evolución del código.

A diferencia de estos lenguajes como Java o C++, en Python no hay un mecanismo para ocultar completamente los atributos o métodos de una clase.

Sin embargo, se puede simular el encapsulamiento en Python mediante convenciones de nomenclatura. En Python, se considera que los atributos o métodos que comienzan con un guión bajo (por ejemplo, _atributo) son de uso interno de la clase y no deben ser accedidos desde fuera de la misma.

Modificadores de acceso

A diferencia de otros lenguajes de programación como Java, C++ o C#, Python no tiene modificadores de acceso explícitos, sino que se basa en convenciones de nomenclatura para indicar la visibilidad de los miembros de una clase

La convención más comúnmente utilizada en Python es la siguiente:

- Atributos y métodos que comienzan con un guión bajo (_) se consideran como atributos y métodos protegidos. Estos miembros se pueden acceder desde la propia clase y desde sus subclases, pero no desde fuera de la clase.

- Atributos y métodos que comienzan con dos guiones bajos (__), se consideran como atributos y métodos privados. Estos miembros solo se pueden acceder desde la propia clase, pero no desde sus subclases ni desde fuera de la clase.

- Atributos y métodos que no tienen guiones bajos al inicio se consideran públicos. Estos miembros se pueden acceder desde cualquier lugar, ya sea desde dentro de la clase, desde fuera de la clase o desde sus subclases.

```python
class Persona:
    def __init__(self, nombre, edad, email):
        self._nombre = nombre    # atributo protegido
        self.__edad = edad       # atributo privado
        self.email = email       # atributo público

    def get_edad(self):
        return self.__edad      # método privado

    def set_edad(self, edad):
        if edad >= 0:
            self.__edad = edad  # método privado

    def saludo(self):
        print("Hola, mi nombre es", self._nombre)
```

Herencia en Python

Definición de herencia

La herencia es un mecanismo de reusabilidad y extensibilidad que permite definir nuevas clases a partir de otras ya existentes y ampliar sus capacidades añadiendo nuevas características, atributos y métodos, en la clase derivada si es necesario. Podemos decir que una clase derivada es una clase más especializada.

La herencia ofrece automáticamente la reutilización del código ya que la clase derivada dispone de todo lo que ofrece la clase base.

```python
class Animal:
    def __init__(self, name, age):
        self.name = name
        self.age = age

    def make_sound(self):
        pass

class Dog(Animal):
    def __init__(self, name, age, breed):
        super().__init__(name, age)
        self.breed = breed

    def make_sound(self):
        return "Woof!"

class Cat(Animal):
    def __init__(self, name, age, color):
        super().__init__(name, age)
        self.color = color

    def make_sound(self):
        return "Meow!"
```

Herencia múltiple.

La herencia múltiple en Python permite a una subclase heredar atributos y métodos de múltiples superclases. La sintaxis para definir una clase que hereda de múltiples superclases es la siguiente:

```python
class Subclase(Superclase1, Superclase2, ..., SuperclaseN):
    def __init__(self, arg1, arg2, ..., argN):
        # Llamada a los constructores de las superclases
        Superclase1.__init__(self, arg1)
        Superclase2.__init__(self, arg2)
        ...
        SuperclaseN.__init__(self, argN)
```

Ejemplo:

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

class Corredor(Atleta):
    def correr(self):
        print(f"{self.nombre} está corriendo")

class Nadador(Atleta):
    def nadar(self):
        print(f"{self.nombre} está nadando")

class AtletaCompleto(Corredor, Nadador):
    def __init__(self, nombre, edad):
        super().__init__(nombre, edad)
```

Polimorfismo

El polimorfismo es un concepto clave en la programación orientada a objetos (POO), que permite que los objetos de diferentes clases puedan ser accedidos utilizando el mismo interfaz, mostrando un comportamiento distinto según cómo sean accedidos [0]. El término polimorfismo significa "muchas formas" y se refiere a que los objetos pueden tomar diferentes formas [1]. En Python, el polimorfismo se ve desde el punto de vista del duck typing, que es la capacidad del intérprete de Python para determinar si un objeto es compatible con una operación en tiempo de ejecución, en lugar de en tiempo de compilación.

```python
def imprimir_datos(nombre, *args, **kwargs):
    print(f"Nombre: {nombre}")

    for arg in args:
        print(f"Valor adicional: {arg}")
    for clave, valor in kwargs.items():
        print(f"{clave}: {valor}")
    print(type(kwargs))

imprimir_datos("Juan", "apellido", "edad", ciudad="Madrid", pais="España")

imprimir_datos("Pedro", "apellido", "edad", ciudad="Madrid", )
```

¿Cuántos tipos de polimorfismo hay en POO...?

En la Programación Orientada a Objetos (POO), hay dos tipos de polimorfismo: el polimorfismo de enlace estático y el polimorfismo de enlace dinámico.

El polimorfismo de enlace estático se logra mediante la sobrecarga de métodos, que es cuando una clase tiene varios métodos con el mismo nombre pero con diferentes parámetros.

```python
class Sobrecarga2:
    def __init__(self, *valores) -> None: ##argumentos *args
        self.valor = valores
        pass

    def suma(self):
        self.sum = 0

        for numero in self.valor:
            self.sum = self.sum + numero
        print("Suma:", self.sum)

sob = Sobrecarga2(1)
print(sob.suma())
```

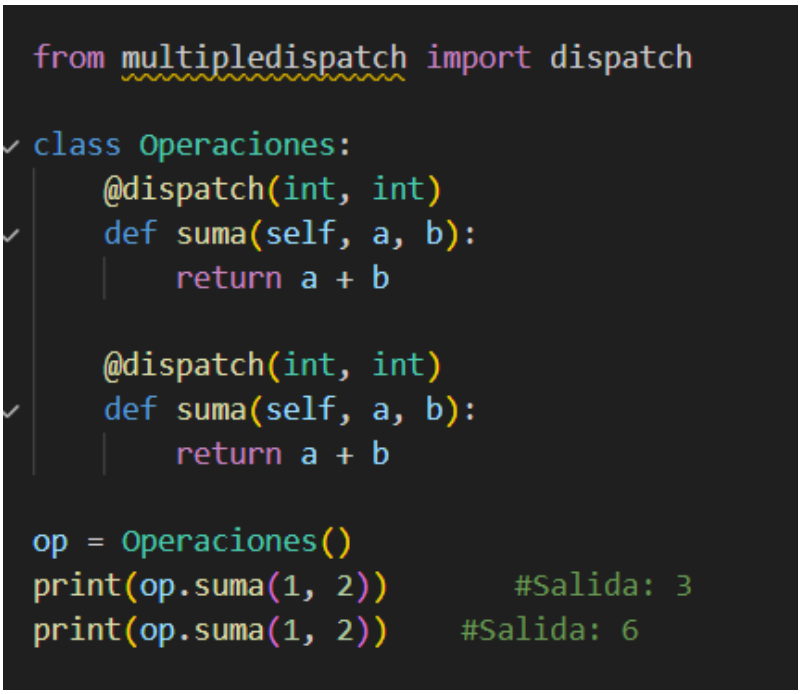
by: Juan Francisco

```python
def imprimir_datos(nombre, *args, **kwargs):
    print(f"Nombre: {nombre}")

    for arg in args:
        print(f"Valor adicional: {arg}")
    for clave, valor in kwargs.items():
        print(f"{clave}: {valor}")
    print(type(kwargs))

imprimir_datos("Juan", "apellido", "edad", ciudad="Madrid", pais="España")

imprimir_datos("Pedro", "apellido", "edad", ciudad="Madrid", )
```

También tenemos la opción que es por medio de decoradores.

En Python, no se puede realizar sobrecarga de métodos directamente, pero hay varias formas de lograrlo. Una forma de realizar la sobrecarga de métodos en Python es utilizando un número variable de argumentos para permitir que un método acepte diferentes tipos de argumentos [4].

La sobrecarga de métodos en Python utilizando el módulo multipledispatch. Primero, se debe instalar el módulo utilizando el comando "pip3 install multipledispatch" en la terminal. Luego, se pueden crear varias funciones con el mismo nombre y agregar un decorador de función justo encima de la función. El decorador de función se utiliza para especificar la función que se utilizará en función de los argumentos proporcionados. Por ejemplo:



Por otro lado, el polimorfismo de sobrescritura (o sobrescritura de método): se refiere a la capacidad de una clase hija de redefinir un método de su clase padre con su propia implementación.

```python
class Vehiculo:
    def conducir(self):
        print("El vehículo está en movimiento")

class Auto(Vehiculo):
    def conducir(self):
        print("El auto está en movimiento")

vehiculo = Vehiculo()
auto = Auto()
vehiculo.conducir() # Imprime "El vehículo está en movimiento"
auto.conducir() # Imprime "El auto está en movimiento"
```

Relaciones entre clases

Las relaciones entre clases en la programación orientada a objetos (POO) son fundamentales para modelar adecuadamente la estructura y el comportamiento de un sistema. Estas relaciones permiten establecer la interacción y la dependencia entre los objetos del sistema, lo que a su vez facilita la creación de un diseño modular, flexible y fácil de mantener. Aquí te detallo el por qué de cada una de las relaciones:

Existen varios tipos de relaciones entre clases en la POO:

1. **Asociación**: La asociación es una relación débil entre dos clases en la que una clase utiliza objetos de la otra clase. Puede ser una asociación unidireccional o bidireccional. Esta relación se establece cuando una clase necesita acceder a los objetos de otra clase para realizar alguna acción, pero no hay una dependencia fuerte entre ellas. Por ejemplo, una clase "Estudiante" puede tener una asociación con una clase "Universidad", donde un estudiante puede estar asociado a una universidad, pero la universidad no depende directamente de los estudiantes.

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

    def __str__(self) -> str:
        return f"{self.nombre}"

class Estudiante:
    def __init__(self, nombre, profesor):
        self.nombre = nombre
        self.profesor = profesor

    def __str__(self) -> str:
        return f"Estudiante: {self.nombre} y nombre del profesor: {self.profesor}"
```

2. **Agregación**: La agregación es una relación en la que una clase contiene una colección de objetos de otra clase. La clase contenedora tiene una referencia a los objetos contenidos, pero los objetos contenidos pueden existir independientemente de la clase contenedora. Por ejemplo, una clase "Equipo" puede tener una agregación con la clase "Jugador", donde un equipo contiene varios jugadores. Un jugador puede pertenecer a un equipo, pero también puede existir fuera de él.

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

class Departamento:
    def __init__(self, nombre):
        self.nombre = nombre
        self.empleados = []

    def agregar_empleado(self, empleado):
        self.empleados.append(empleado)

empleado1 = Empleado("Juan")
empleado2 = Empleado("Ana")
departamento = Departamento("Recursos Humanos")
departamento.agregar_empleado(empleado1)
departamento.agregar_empleado(empleado2)
```

3. **Composición**: La composición es una relación más fuerte que la agregación. En la composición, una clase está compuesta por objetos de otra clase, y los objetos componentes no pueden existir sin la clase compuesta. La clase compuesta es responsable de la creación y destrucción de los objetos componentes. Por ejemplo, una clase "Coche" puede tener una composición con la clase "Motor", donde un coche está compuesto por un único motor. Si el coche se destruye, el motor también se destruye.

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

class Coche:
    def __init__(self, marca, modelo, anio, neumaticos):
        self.marca = marca
        self.modelo = modelo
        self.anio = anio
        self.neumaticos = [Neumatico(marca) for i in range(neumaticos)]

coche = Coche("Ford", "Mustang", 2022, 4)
```

4. **Herencia**: La herencia es una relación en la que una clase hereda los atributos y métodos de otra clase. La clase que hereda se denomina clase derivada o subclase, y la clase de la que hereda se denomina clase base o superclase. La herencia permite la reutilización de código y la especialización de comportamiento. La subclase puede agregar nuevos atributos o métodos, o sobrescribir los existentes de la superclase. Por ejemplo, una clase "Gato" puede heredar de una clase base "Animal", obteniendo así los atributos y métodos comunes a todos los animales, pero también puede tener atributos y métodos específicos de los gatos.

Estas relaciones entre clases son esenciales en la POO para modelar la estructura y el comportamiento de un sistema de manera modular y flexible. Cada relación tiene sus propias características y niveles de dependencia entre las clases involucradas, lo que permite representar de manera más precisa las interacciones y dependencias entre los objetos del sistema.

Excepciones.

Una excepción es la indicación de que se produjo un error en el programa. Las excepciones, como su nombre lo indica, se produce cuando la ejecución de un método no termina correctamente, si no que termina de manera excepcional como consecuencia de una situación no esperada.

Cuando se produce una situación anormal durante la ejecución de un programa (por ejemplo, se accede a un objeto que no ha sido inicializado o tratamos de acceder a una posición inválida en un vector), si no manejamos de manera adecuada el error que se produce, el programa va a terminar abruptamente su ejecución. Decimos que el programa deja de funcionar y es muy probable que el usuario que lo estaba utilizando ni siquiera sepa qué fue lo que pasó.

Durante la ejecución de un programa, si dentro de una función surge una excepción y la función no lo maneja, la función se propaga a la función que lo invocó, si esta otra tampoco lo maneja, la excepción continúa propagándose hasta llegar a la función inicial del programa y si esta tampoco lo maneja se interrumpí la ejecución del programa.

Manejo de excepciones.

Para el manejo de excepciones los lenguajes proveen ciertas palabras reservadas, que nos permiten manejar las excepciones que puedan surgir y tomar acciones de recuperación para evitar la interrupción del programa o, al menos, para realizar algunas acciones antes de interrumpir el programa.
En el caso de Python el manejo de excepciones se hace mediante los bloques que utilizan las sentencias *try, except y finally.*

Dentro del bloque *try* se ubica todo el código que pueda llegar a levantar una excepción. A continuación se ubica el bloque *except*, que se encarga de capturar la excepción y nos da la oportunidad de procesarla mostrando, por ejemplo, un mensaje adecuado al usuario.

Ejemplo de una división entre cero:

```
dividendo = 5
divisor = 0
print(dividendo/divisor)
```

```
C:\Windows\System32\cmd.exe
Microsoft Windows [Versión 10.0.19045.3086]
(c) Microsoft Corporation. Todos los derechos reservados.

C:\Users\Sertwo\Downloads>python sis_universitario.py
Traceback (most recent call last):
  File "C:\Users\Sertwo\Downloads\sis_universitario.py", line 104, in <module>
    print(dividendo/divisor)
          ~~~~~~~~~~^~~~~~~~
ZeroDivisionError: division by zero

C:\Users\Sertwo\Downloads>
```

En este caso, se levantó la excepción ZeroDivisionError cuando se quiso hacer la división. Para evitar que se levante la excepción y se detenga la ejecución del programa, se utiliza el bloque *try-except.*

```
try:
    # Es un lugar donde
    # tu puedes hacer algo
    # sin pedir permiso.
except:
    # Es un espacio dedicado
    # exclusivamente para pedir perdón.
```

Entonces, podríamos decir que estos dos bloques funcionan así:

- la palabra clave reservada *try* marca el lugar donde intentas hacer algo sin permiso;
- la palabra clave reservada *except* comienza un lugar donde puedes mostrar tu talento para disculparte o pedir perdón.

```
try:
    dividendo = 5
    divisor = 0
    print(dividendo/divisor)
except:
    print("No se permite la división entre cero")
```

```
C:\Windows\System32\cmd.exe

C:\Users\Sertwo\Downloads>python sis_universitario.py
No se permite la división entre cero

C:\Users\Sertwo\Downloads>
```

Dos excepciones después de un try

Observa el siguiente código. Como se puede ver, acabamos de agregar un segundo *except*. Esta no es la única diferencia (toma en cuenta que ambos *except* tiene nombres de excepciones específicos). En esta variante, cada una de las excepciones esperadas tiene su propia forma de manejar el error, pero debe enfatizarse en que solo una de todas puede interceptar el control -si se ejecuta una, todas las demás quedan inactivas.

```
try:
    value = int(input('Ingresa un número natural: '))
    print('El recíproco de', value, 'es', 1/value)
except ValueError:
    print('No se que hacer con', value)
except ZeroDivisionError:
    print('La división entre cero no está permitida en nuestro Universo.')
```

Además la cantidad de *except* no está limitada - puedes especificar tanta o tan pocas como necesites, pero no se te olvide que **ninguna de las excepciones se puede especificar más de una vez.**

Cohesión y Acoplamiento

El acoplamiento y la cohesión son dos conceptos clave en la Ingeniería de software que se utilizan para medir la calidad del diseño de un sistema de software.

Representación gráfica de la alta cohesión y el bajo acoplamiento

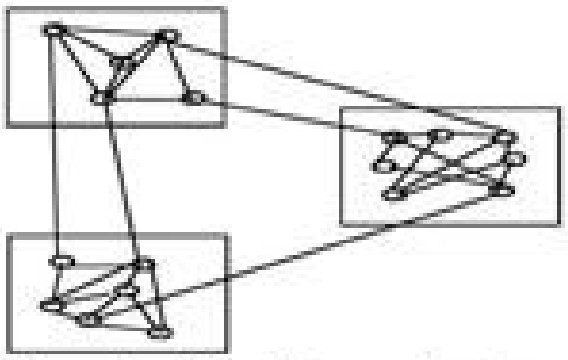


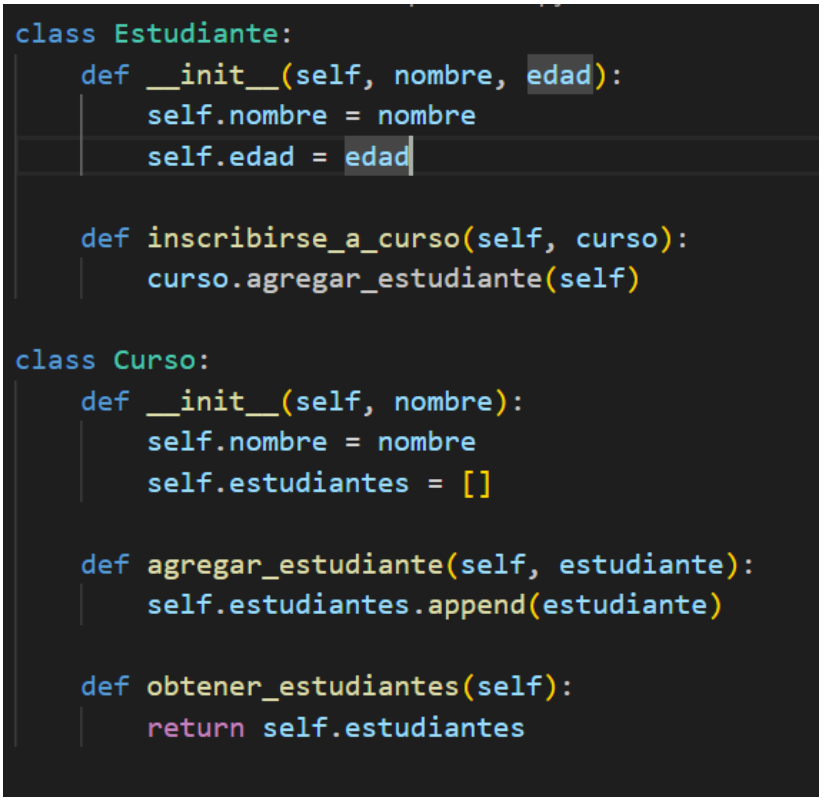
Figura 4. Alta cohesión y bajo acoplamiento [4]

¿Qué es la cohesión?

La cohesión se refiere al grado en que los elementos dentro de un módulo están relacionados entre sí. Una alta cohesión significa que los elementos dentro del módulo están altamente relacionados, lo que significa que el módulo se centrará en una sola tarea y lo hará bien. La alta cohesión nos permitirá reutilizar las clases y métodos.

¿Qué es el acoplamiento?

El acoplamiento se refiere al grado de interdependencia entre los módulos de software. El acoplamiento alto significa que los módulos están estrechamente conectados y los cambios en un módulo pueden afectar a otros módulos. El acoplamiento bajo significa que los módulos son independientes y los cambios en un módulo tienen poco impacto en otros módulos.



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

    def inscribirse_a_curso(self, curso):
        curso.agregar_estudiante(self)

class Curso:
    def __init__(self, nombre):
        self.nombre = nombre
        self.estudiantes = []

    def agregar_estudiante(self, estudiante):
        self.estudiantes.append(estudiante)

    def obtener_estudiantes(self):
        return self.estudiantes
```

Desventajas del acoplamiento alto:

- Mayor complejidad: el acoplamiento alto aumenta la interdependencia entre módulos, lo que hace que el sistema sea más complejo y difícil de entender.
- Flexibilidad reducida: el acoplamiento alto dificulta la modificación o el reemplazo de componentes individuales sin afectar a todo el sistema.
- Disminución de la modularidad: el acoplamiento alto dificulta el desarrollo y prueba de módulos de forma aislada, reduciendo la modularidad y la reutilización del código.

Desventajas de la baja cohesión:

- Mayor duplicación de código: la baja cohesión puede conducir a la duplicación de código, ya que los elementos que pertenecen juntos se dividen en módulos separados.
- Funcionalidad reducida: la baja cohesión puede dar como resultado módulos que carecen de un propósito claro y contienen elementos que no pertenecen juntos, lo que reduce su funcionalidad y los hace más difíciles de mantener.
- Dificultad para comprender el módulo: la baja cohesión puede dificultar a los desarrolladores comprender el propósito y el comportamiento de un módulo, lo que genera errores y una falta de claridad

Ventajas del acoplamiento bajo:

- Mejora de la mantenibilidad: el acoplamiento bajo reduce el impacto de los cambios en un módulo en otros módulos, lo que facilita la modificación o reemplazo de componentes individuales sin afectar a todo el sistema.
- Modularidad mejorada: el acoplamiento bajo permite desarrollar y probar módulos de forma aislada, mejorando la modularidad y la reutilización del código.
- Mejor escalabilidad: el acoplamiento bajo facilita la adición de nuevos módulos y la eliminación de los existentes, lo que facilita la escala del sistema según sea necesario.

Ventajas de la alta cohesión:

- Legibilidad y comprensibilidad mejoradas: la alta cohesión da como resultado módulos claros y enfocados con un único propósito bien definido, lo que facilita a los desarrolladores comprender el código y realizar cambios.
- Mejor aislamiento de errores: la alta cohesión reduce la probabilidad de que un cambio en una parte de un módulo afecte a otras partes, lo que facilita aislar y corregir errores.
- Fiabilidad mejorada: la alta cohesión conduce a módulos que son menos propensos a errores y que funcionan de manera más consistente, que conduce a una mejora general en la confiabilidad del sistema.

```python
class Persona:
    def __init__(self, nombre, apellido) -> None:
        self.id = randint(1,100)
        self.nombre = nombre
        self.apellido = apellido

    def  __str__(self) -> str:
        return f"{self.__class__.__name__}: {self.id} {self.nombre} {self.apellido}"

class Curso:
    def __init__(self, nombre) -> None:
        self.nombre = nombre

    def __str__(self) -> str:
        return self.nombre

class Estudiante(Persona):
    def __init__(self, nombre, apellido) -> None:
        super().__init__(nombre, apellido)
        fecha = datetime.now() #fecha es una variable local  mas no de instancia
        self.fecha_registro = fecha.strftime("%Y-%m-%d %H:%M")
        self.cursos = [] #Agregación

    def __str__(self) -> str:
        cursos_str = ', '.join(str(c) for c in self.cursos)
        return super().__str__() + " " + str(self.fecha_registro) + " " + cursos_str

    def agregar_curso(self, nombre_curso):
        #Relación de Composición
        curso = Curso(nombre_curso) #creamos instacia de un cursdo cada vez que se llame a este método
        self.cursos.append(curso)

class Universidad:
    def __init__(self, nombre) -> None:
        self.nombre = nombre
        self.estudiantes = []
        self.df = pd.DataFrame(columns=["ID", "Nombre", "Apellido", "Fecha_registro", "Cursos"])

    def agregar_estudiante(self, estudiante):
        #Relacion de agregación
        self.estudiantes.append(estudiante)

    def __str__(self) -> str:
        estudiates_str = ', '.join(str(estudiante) for estudiante in self.estudiantes)#estudiates_str es una variable local y no una de instancia
        return f"{self.nombre}\n{estudiates_str}"
```

No hay que abusar:

```python
class SquareCalculator:
    def __init__(self, number):
        self.number = number

    def calculate_square(self):
        return self.number ** 2

class SineCalculator:
    def __init__(self, number):
        self.number = number

    def calculate_sine(self):
        # Cálculo del seno del número
        pass

class CosCalculator:
    def __init__(self, number):
        self.number = number

    def calculate_cos(self):
        # Cálculo del seno del número
        pass

class FactorialCalculator:
    def __init__(self, number):
        self.number = number

    def calculate_factorial(self):
        # Cálculo del factorial del número
        pass
```

Clases Abstractas

Las clases abstractas son clases que no pueden ser instanciadas y que se utilizan para definir una estructura o modelo a seguir por las clases hijas que la heredan. En Python, para definir una clase abstracta se utiliza el módulo "abc" y la clase "ABC" como base abstracta. Estas clases pueden contener métodos abstractos, que son métodos que deben ser implementados por las clases hijas.

```python
from abc import ABC, abstractmethod

class FiguraGeometrica(ABC):

    @abstractmethod
    def area(self):
        pass

    @abstractmethod
    def perimetro(self):
        pass

class Cuadrado(FiguraGeometrica):

    def __init__(self, lado):
        self.lado = lado

    def area(self):
        return self.lado ** 2

    def perimetro(self):
        return self.lado * 4
```

Principios S.O.L.I.D.

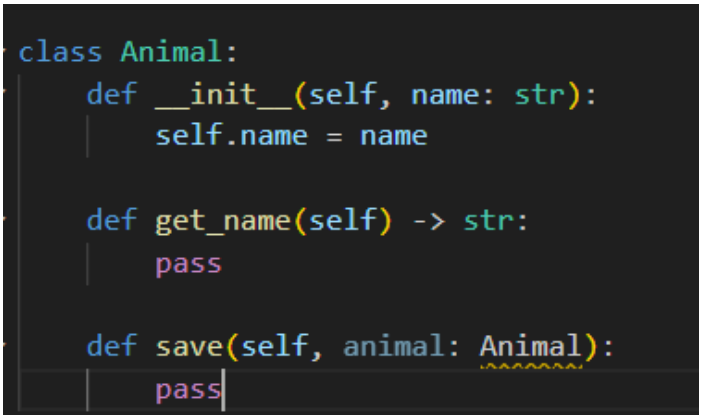
Los Principios SÓLIDOS son cinco principios del diseño de clase orientado a objetos. Son un conjunto de reglas y mejores prácticas a seguir mientras diseñan una estructura de clase. Estos cinco principios nos ayudan a comprender la necesidad de ciertos patrones de diseño y arquitectura de software en general.

Estos principios establecen prácticas que se prestan al desarrollo de software con consideraciones para mantener y extender a medida que el proyecto crece. La adopción de estas prácticas también puede contribuir a evitar errores de código, refactorizar código y desarrollar software ágil o adaptativo.

Estos principios son un acrónimo de los cinco principios que se detallan a continuación:

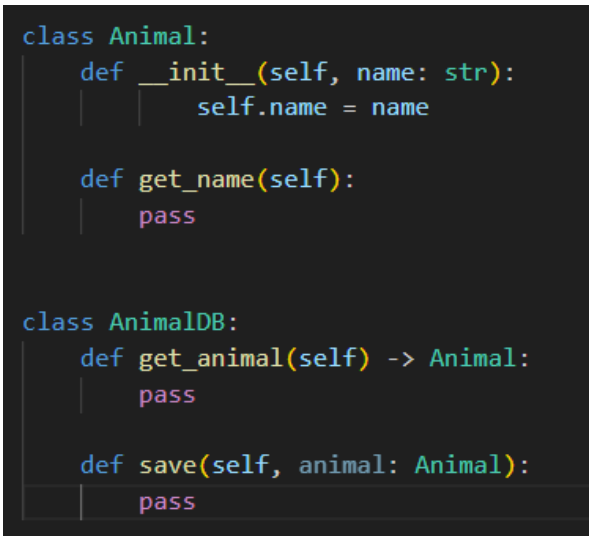
Principio de Responsabilidad única

Un componente de software debe tener una sola responsabilidad. Debe tener solo una razón para cambiar. Si necesita modificar la clase por diferentes razones, esto significa que falta algo (una abstracción) y debe solucionarlo. Por ejemplo, cuando observa los métodos de la clase y ve que hay algunas funciones no relacionadas (entre sí), puede decir que esta clase tiene que desglosarse.



```python
class Animal:
    def __init__(self, name: str):
        self.name = name

    def get_name(self) -> str:
        pass

    def save(self, animal: Animal):
        pass
```

Aplicando el principio de Responsabilidad Única:



```python
class Animal:
    def __init__(self, name: str):
        self.name = name

    def get_name(self):
        pass


class AnimalDB:
    def get_animal(self) -> Animal:
        pass

    def save(self, animal: Animal):
        pass
```

Principio de Abierto/Cerrado:

Una clase debe estar abierta a extensión pero cerrada a modificación. Cuando queremos agregar cosas nuevas a nuestro modelo, sólo queremos agregar cosas nuevas y no cambiar nada existente, y que esté cerrado a modificaciones. Este principio es muy fácil de entender con un ejemplo concreto. Comencemos con un mal ejemplo:

```python
class Animal:
    def __init__(self, name: str):
        self.name = name

    def get_name(self) -> str:
        pass

animals = [
    Animal('lion'),
    Animal('mouse')
]

def animal_sonido(animals: list):
    for animal in animals:
        if animal.name == 'lion':
            print('roar')

        elif animal.name == 'mouse':
            print('squeak')

animal_sonido(animals)
```

¿Qué pasa si queremos agregarle más animales?

```python
animals = [
    Animal('lion'),
    Animal('mouse'),
    Animal('snake')
]

def animal_sound(animals: list):
    for animal in animals:
        if animal.name == 'lion':
            print('roar')
        elif animal.name == 'mouse':
            print('squeak')
        elif animal.name == 'snake':
            print('hiss')

animal_sound(animals)
```

Aplicando el principio de Abierto/Cerrado:

```python
class Animal:
    def __init__(self, name: str):
        self.name = name

    def get_name(self) -> str:
        pass

    def make_sound(self):
        pass

class Lion(Animal):
    def make_sound(self):
        return 'roar'

class Mouse(Animal):
    def make_sound(self):
        return 'squeak'

class Snake(Animal):
    def make_sound(self):
        return 'hiss'

def animal_sound(animals: list):
    for animal in animals:
        print(animal.make_sound())

animal_sound(animals)
```

Principio de Sustitución de Liskov:

La L de SOLID alude al apellido de quien lo creó, Barbara Liskov, y dice que “las clases derivadas deben poder sustituirse por sus clases base”.

Esto significa que los objetos deben poder ser reemplazados por instancias de sus subtipos sin alterar el correcto funcionamiento del sistema o lo que es lo mismo: si en un programa utilizamos cierta clase, deberíamos poder usar cualquiera de sus subclases sin interferir en la funcionalidad del programa.

```python
class Animal:
    def comer(self):
        print("El animal está comiendo")

class Perro(Animal):
    def comer(self):
        print("El perro está comiendo")

class Gato(Animal):
    def comer(self):
        print("El gato está comiendo")

def alimentar_animal(animal):
    animal.comer()

perro = Perro()
gato = Gato()

alimentar_animal(perro) # salida: "El perro está comiendo"
alimentar_animal(gato) # salida: "El gato está comiendo"
```

Principio de Segregación de Interfaces

El principio de segregación de la interfaz establece que una interfaz debe ser lo más pequeña posible en términos de cohesión. En otras palabras, debería hacer UNA cosa. No significa que la interfaz deba tener un método. Una interfaz puede tener múltiples métodos cohesivos.

Antes:

```python
from abc import ABC, abstractmethod

class Vehiculo(ABC):
    @abstractmethod
    def andar(self):
        pass

    @abstractmethod
    def volar(self):
        pass

class Aeronave(Vehiculo):
    def andar(self):
        print("Rodando")

    def volar(self):
        print("Volando")

class Carro(Vehiculo):
    def andar(self):
        print("Andando")

    def volar(self):
        raise Exception('El carro no puede volar')
```

Después:

```python
class Vehiculo(ABC):
    @abstractmethod
    def andar(self):
        pass

class Volador(ABC):
    @abstractmethod
    def volar(self):
        pass

class Aeronave(Volador, Vehiculo):
    def andar(self):
        print("Rodando")

    def volar(self):
        print("Volando")

class Carro(Vehiculo):
    def andar(self):
        print("Andando")
```

Inversión de Dependencias:

El principio de inversión de dependencias establece que los módulos de nivel superior no deben depender directamente de los módulos de nivel inferior , sino que ambas clases deben depender de abstracciones. Además, las abstracciones no deben depender de los detalles, sino que los detalles deben depender de abstracciones.

Ejemplo:

```python
from abc import ABC, abstractmethod

class Motor(ABC):

    @abstractmethod
    def encender(self):
        pass

    @abstractmethod
    def apagar(self):
        pass

class Coche:

    def __init__(self, motor: Motor):
        self.motor = motor

    def encender_coche(self):
        self.motor.encender()

    def apagar_coche(self):
        self.motor.apagar()

class MotorGasolina(Motor):

    def encender(self):
        print("Encendiendo motor de gasolina...")

    def apagar(self):
        print("Apagando motor de gasolina...")

class MotorElectrico(Motor):

    def encender(self):
        print("Encendiendo motor eléctrico...")

    def apagar(self):
        print("Apagando motor eléctrico...")
```

Referencias:

http://www.dynadata.com/ITVER/Docs/Simulacion/UNIDAD%203%20HERRAMIENTAS%20DE%20PROGRAMACION/Introduccion%20a%20la%20Programacion%20Orientada%20a%20Objetos.pdf

https://d1wqtxts1xzle7.cloudfront.net/36832260/Programacion_Orientada_a_Objeto_Conceptos_Basico-libre.pdf?1425339059=&response-content-disposition=inline%3B+filename%3DUniversidad_Central_de_Venezuela_Tema_8.pdf&Expires=1681003442&Signature=JxswMT5kE5e7eDoGdH11GEpTikvg78sKXbNSFlJ3LbodHuwWYNvvGpze4Nbw3iBoVQq--gXwYcKuIflA8VDSnN0f18SqHkqo09~MxajyuEhWtpl7xlaxZbO~N3LjuJGPc9HF4GGufRo4vj40ffCIqk4G~suQX0VgMtfjI6oVLkikdIgEJh0tZVF~ryh-OoZAcqUIYPPDBEQMuqZUmsDMfzRElCg~ZzblZt2VmAfXWuynF1sOzr-1at6YH-1MhHgSzReyGbAEyOx8Upv1muPJXiScbm8GaNSSv8MD-5NXxh4d4TomU1YMiqnD5pSMP0fTvzadT-u0mn7PcWkdyJNCnA__&Key-Pair-Id=APKAJLOHF5GGSLRBV4ZA

https://www.tutorialspoint.com/python/python_classes_objects.htm

https://gamedevtraum.com/es/programacion-informatica/programacion-orientada-a-objetos/diferencia-entre-clase-y-objeto/#:~:text=Una%20clase%20es%20una%20estructura,cuando%20el%20programa%20est%C3%A1%20corriendo.

https://realpython.com/python3-object-oriented-programming/#what-is-object-oriented-programming-in-python

https://peps.python.org/pep-0008/#tabs-or-spaces

http://www.tugurium.com/python/index.php?C=PYTHON.11

https://elpythonista.com/zen-de-python
https://ellibrodepython.com/polimorfismo-en-programacion

https://www.tutorialspoint.com/python/python_exceptions.htm

https://www.geeksforgeeks.org/software-engineering-coupling-and-cohesion/

https://www.pythontutorial.net/python-oop/python-single-responsibility-principle/

https://www.linkedin.com/pulse/solid-principles-tutorial-java-coding-example-beginners-digest

https://profile.es/blog/principios-solid-desarrollo-software-calidad/