# Algorítmia y lenguaje python orientado a objetos con 20 ejemplos prácticos

MSc. Luis Fdo Alvarez V.
MSc. Leonardo Alvarez V.
MSc. Hernando Alvarez R.

Universidad Tecnológica
de Pereira

# Prefacio

El mundo de la programación y el diseño de algoritmos ha evolucionado a un ritmo vertiginoso, exigiendo a estudiantes, profesionales y entusiastas mantenerse actualizados con herramientas y enfoques modernos. “Algoritmia y lenguaje python orientado a objetos con 20 ejemplos prácticos” nace como un esfuerzo por brindar una guía clara y completa que combina los fundamentos de la algoritmia con el poder y la flexibilidad del lenguaje python, todo enmarcado en los principios de la programación orientada a objetos (POO).

El enfoque que se presenta, es eminentemente práctico, diseñado para guiar al lector desde los conceptos básicos de los algoritmos y el diseño mediante diagramas Nassi-Schneiderman (N-S), hasta la implementación de soluciones robustas y elegantes en python. Cada capítulo ha sido estructurado con el propósito de facilitar un aprendizaje progresivo, donde la teoría se integra con ejercicios y ejemplos que reflejan situaciones reales y relevantes en el ámbito de la tecnología y las ciencias computacionales.

Para facilitar el aprendizaje progresivo, hemos estructurado su contenido en capítulos que desarrollan conceptos claves en torno a la lógica computacional, el diseño de algoritmos y su implementación en lenguaje python. A continuación, se presenta una breve descripción del contenido de cada uno de ellos, con el fin de ofrecer al lector una visión general del recorrido formativo que encontrará a lo largo del texto.

**Capítulo 1:** Diseño de algoritmos con diagramas N-S. El primer capítulo introduce el diseño algorítmico mediante diagramas N-S, una herramienta visual poderosa para estructurar ideas y resolver problemas de manera sistemática. Este enfoque prepara al lector para pensar de forma lógica y clara antes de abordar la implementación en código. Para más detalles sobre los diagramas de Nassi - Shneiderman

**Capítulo 2:** Iniciando con el lenguaje python. Reconocido por su sintaxis sencilla y versatilidad, python es el lenguaje elegido para este libro. En este capítulo, el lector aprenderá los fundamentos del lenguaje, desde las variables y tipos de datos hasta las estructuras de control y funciones, sentando una base sólida para los temas más avanzados.

**Capítulo 3:** Programación orientada a objetos con python. Este capítulo explora los conceptos clave de la POO, como clases, objetos, herencia, polimorfismo y encapsulamiento, con un enfoque práctico en su implementación en python. La POO permite estructurar el código de forma más modular. Además, adoptaremos la Metodología Orientada a Objetos (MAOO), desarrollada por Ricardo de Jesús Botero Tabares. Para una comprensión más profunda de esta metodología, te invitamos a consultar su trabajo detallado disponible en la biblioteca digital de la Universidad de Antioquia

**Capítulo 4:** Algoritmia orientada a objetos con UML. La utilización de diagramas UML complementa la enseñanza de la POO, proporcionando herramientas visuales para diseñar e interpretar sistemas complejos. Este capítulo vincula la teoría de la POO con la práctica mediante el uso de diagramas de clases, secuencia y actividad.

**Capítulo 5:** Veinte ejemplos con algoritmia y python orientados a objetos. El libro culmina con veinte ejemplos prácticos que combinan todo lo aprendido. Estos ejercicios abordan problemas reales y desafiantes, desde aplicaciones matemáticas hasta simulaciones y soluciones empresariales, ofreciendo al lector una experiencia completa de aprendizaje.

Este libro es el resultado de un profundo interés por compartir conocimientos de manera accesible y efectiva. Su estructura está pensada para que tanto principiantes como programadores con experiencia puedan beneficiarse y encontrar inspiración en su contenido. Esperamos que esta obra no solo sirva como una herramienta de aprendizaje, sino también como un puente para descubrir el fascinante mundo de la algoritmia y la programación orientada a objetos.

¡Bienvenidos a este emocionante viaje hacia el dominio de la algoritmia y python!

# Índice general

# 1 Diseño de algoritmos con diagramas N-S

## 1.1 Diseño de algoritmos

**Algoritmo:** Conjunto ordenado y finito de operaciones que permiten la solución de un problema en general. Los algoritmos son el fundamento de la programación de computadores.

Dos características muy importante en el diseño de algoritmos:

- Ser preciso: las operaciones o pasos del algoritmo deben desarrollarse en orden estricto ya que el desarrollo de cada paso debe obedecer a un orden lógico
- Ser definido: ya que en el área de programación el algoritmo se elabora como paso fundamental para escribir el programa

### 1.1.1 Pasos para el diseño de algoritmos

- Paso 1: Conocer la temática a tratar.
- Paso 2: Actividades a realizar.
- Paso 3: Presentación formal del algoritmo

### 1.1.2 Formas de presentación de algoritmos

Entre las más destacadas, podemos mencionar:

- Pseudocódigo
- Diagramas de flujo
- Diagramas Nassi - Schniederman (N-S)

## 1.2 Introducción a los diagramas Nassi-Shneiderman (N-S)

El diseño de algoritmos es una de las habilidades fundamentales en programación. Antes de escribir el código, es crucial tener una visión clara del problema que se desea resolver y la

lógica que se empleará para lograrlo. Un forma eficiente y visual de representar esta lógica es a través de los diagramas N-S, los cuales se basan en pseudocódigo y hacen uso de tres figuras que corresponden a las estructuras de programación básicas (secuenciales, condicionales, repetitivas). Una de sus características distintivas radica en que no emplean flechas para mostrar la secuencia de operaciones; en cambio, se utilizan cajas que se disponen una tras otra o se anidan entre sí, lo cual pone de manifiesto la sucesión de pasos, la bifurcación y la repetición del algoritmo

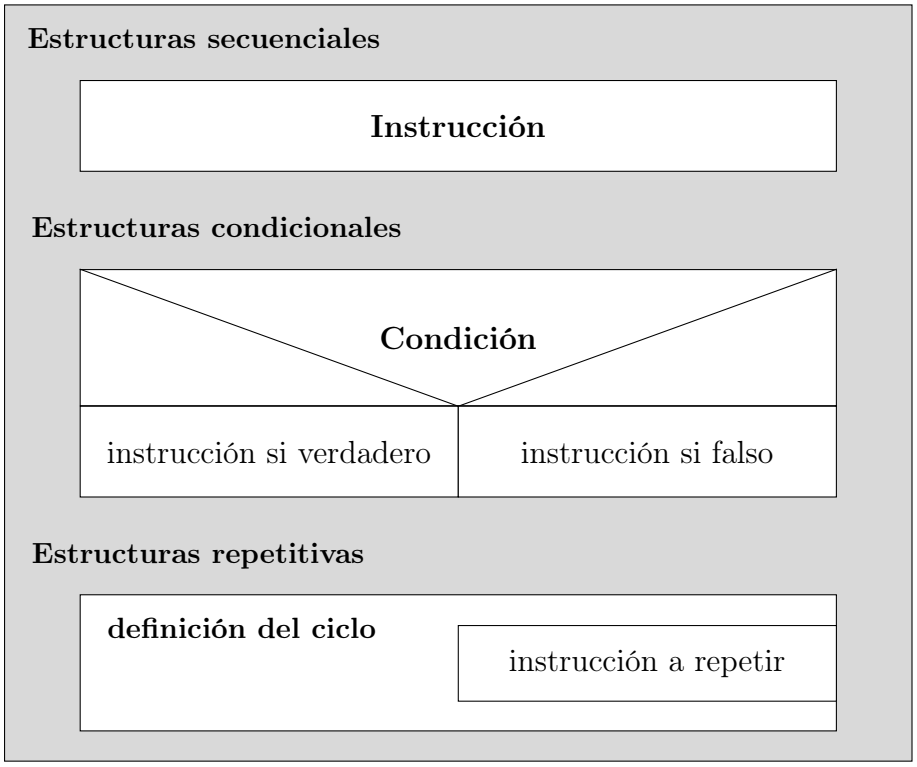



**Figura 1.1:** Representación de las estructuras en los diagramas N-S.

### 1.2.1 Breve historía

Los diagramas N-S fuerón desarrollados en 1972 por Isaac Robert Nassi y Ben Shneiderman como una alternativa a los diagramas de flujo tradicionales. Su diseño estructurado elimina el uso de flechas y otras conexiones confusas, favoreciendo una representación más clara y jerárquica de las estructuras de control como secuencias, condicionales y ciclos. Esto los convierte en una herramienta ideal para diseñar algoritmos de manera eficiente y comprensible.

### 1.2.2 Importancia

Estos diagramas son una poderosa herramienta para:

- Comprender y visualizar algoritmos: Proporcionan una representación clara y estructurada de la lógica de un programa.

- Facilitar la comunicación: Ayudan a programadores, estudiantes y equipos de desarrollo a compartir ideas de manera efectiva
- Prevenir errores de lógica: Al diseñar primero el diagrama, se puede identificar y corregir errores antes de implementarlos.
- Servir como base para la transición a la programación orientada a objetos (POO): En este enfoque, los diagramas N-S se utilizan para diseñar algoritmos que luego se transforman en métodos y clases en python

### 1.2.3 Conceptos básicos

**¿Qué es un diagrama Nassi - Shneiderman?**

Un diagrama Nassi - Shneiderman es una herramienta visual que representa las estructuras de control de un algoritmo (secuencias, condiciones y ciclos) de manera jerárquica y estructurada.

Cada bloque en el diagrama representa una acción o conjunto de acciones y su disposición indica el flujo del programa.

**Características principales**

1. Se basa en la programación estructurada, organizando el código en estructuras bien definidas.
2. Utiliza bloques rectangulares para representar acciones.
3. El flujo del programa se mueve de arriba hacia abajo o de izquierda a derecha, dependiendo del diseño.
4. Evita el uso de elementos confusos como flechas cruzadas o conexiones complejas.

**Ventajas**

1. Claridad visual: Su estructura limpia y jerárquica facilita la comprensión del algoritmo.
2. Facilidad de tradución: Los diagramas N-S pueden convertirse directamente en código en lenguajes de programación estructurados.
3. Prevención de errores: Su diseño estructurado ayuda a identificar errores lógicos antes de la implementación.
4. Compatibilidad con la POO: ayudan a diseñar métodos y clases como bloques independientes, favoreciendo el diseño modular.

**Elementos clave**

Para construir un diagrama Nassi - Shneiderman, es esencial comprender sus elementos básicos:

1. **Palabras claves adaptadas del pseudocódigo.**
   - Inicio, Fin
   - Leer, Escribir
   - Mientras, Fin mientras, Repita, Hasta, Para, Fin para
   - Incrementar, Decrementar, Hacer, Funcion, Procedimiento
   - Entero, Real, Caracter, Cadena
   - Logico, Retornar
2. **Estructuras secuenciales:** Son aquellas que se ejecutan una a continuación de la otra.
   - Declaración de variables

| **Declaración de las variables** | **Comentarios** |
|---|---|
| **Tipo :** identificador | **Tipo** puede ser : **Entero, Cadena o Real**<br>identificador: es el nombre asignado por parte del usuario |

Ejemplos

| | |
|---|---|
| **Entero :** altura | altura: identificador variable de tipo Entero |
| **Cadena :** nombre,direccion | Dos variables de tipo Cadena |
| **Real :** sueldo,descuentos,iva | Tres variables de tipo Real |

   - Asignación: Asignar un valor a una variable equivale a decir que se guarda dicho valor en la posición de memoria reservado para la variable en mención. La forma de realizarlo es como sigue:

$$\textbf{identificador} = \textit{expresión}$$

Donde *expresión* puede ser un valor, conjunto de valores y operadores o por una función

Ejemplos:

| | |
|---|---|
| **altura** = 2.35 | altura, es de tipo Real |
| **edad** = 30 | nombre, es de tipo Entero |

   - Instrucciones Leer, Escribir
     - Instrucción **Leer**: La instrucción **Leer** se utiliza para enviar información desde un dispositivo de entrada de datos hacia la memoria. La forma de utilizarla es la siguiente:

$$\textbf{Leer} <\textit{lista de identificadores de variables}>$$

Ejemplo: **Leer** *x,y* *x,y* son las variables para almacenar los valores

- Instrucción **Escribir** Utilizada para imprimir información sobre el dispositivo. Su forma es:

  **Escribir** <*lista de constantes y variables*>

  Ejemplo: **Escribir** *x,y*

  La instrucción **Escribir** permite enviar datos a un dispositivo de salida. Se debe separar con coma cuando hay mas de una variable. Los mensajes se escriben entre comillas dobles. Cuando una variable se escribe entre comillas se mostrara el identificador

**Ejemplo 1** Calcular la suma de dos números reales, digitados por el usuario.

**Definición del problema**
Calcular la suma de dos números reales

**Análisis del problema**
En este problema solamente es sumar los dos números. Sean:

**num1:** primer número
**num2:** segundo número
**resultado** = num1 + num2

Datos de entrada: num1 y num2 (los dos números)
Datos de salida: resultado
Proceso: resultado = num1 + num2

**Diseño de la solución**

| **Inicio** |
|---|
| **Real** num1,num2,resultado |
| **Leer** num1,num2 |
| resultado = num1 + num2 |
| **Escribir** "La suma es:",resultado |
| **Fin** |

**Ejemplo 2** Calcular el perímetro de un triángulo cualquiera, dadas las longitudes de los tres lados.

**Definición del problema**
Calcular el perímetro de un triángulo cualquiera

**Análisis del problema**
En este problema sumaremos los tres lados. Sean:

**lado1:** primer lado
**lado2:** segundo lado
**lado3:** tercer lado
**perimetro** = lado1 + lado2 + lado3

Datos de entrada: lado1, lado2 y lado3 (los tres lados)
Datos de salida: perimetro
Proceso: perimetro = lado1 + lado2 + lado3

**Diseño de la solución**

| |
|---|
| **Inicio** |
| **Real** lado1,lado2,lado3,perimetro |
| **Leer** lado1,lado2,lado3 |
| perimetro = lado1 + lado2 + lado3 |
| **Escribir** .El perimetro es:",perimetro |
| **Fin** |

3. **Estructuras de selección (decisión).** Las estructuras de selección o decisión permiten que el algoritmo tome decisiones y ejecute u omita algunas instrucciones dependiendo de una condición. Pueder ser simples, compuestas y múltiples

   **Ejemplo 3** Dado un número hallar su valor absoluto

   **Definición del problema**
   Hallar el valor absoluto de un número

   **Análisis del problema**
   Símbolo matemático para el valor absoluto $|\ \ |$. Algunos ejemplos

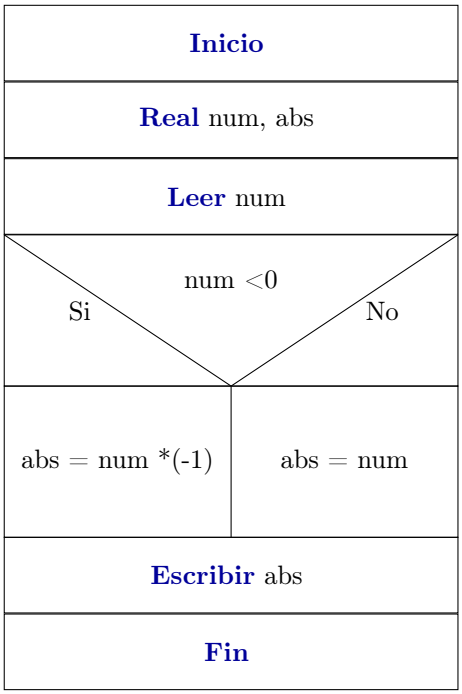
   $$|2| = 2, \qquad |-2| = 2, \qquad |-13| = 13, \qquad |5.19| = 5.19, \qquad |-2.375| = 2.375$$

   $$\text{Definición: } |x| = \begin{cases} x, & \text{si } x \geq 0 \\ -x, & \text{si } x < 0 \end{cases}$$

   En conclusión el valor absoluto de un número es el mismo número, si este es positivo y se multiplica por $(-1)$ si es negativo.

   Datos de entrada: numero
   Datos de salida: abs
   Proceso: abs = numero*(-1)

   **Diseño de la solución**

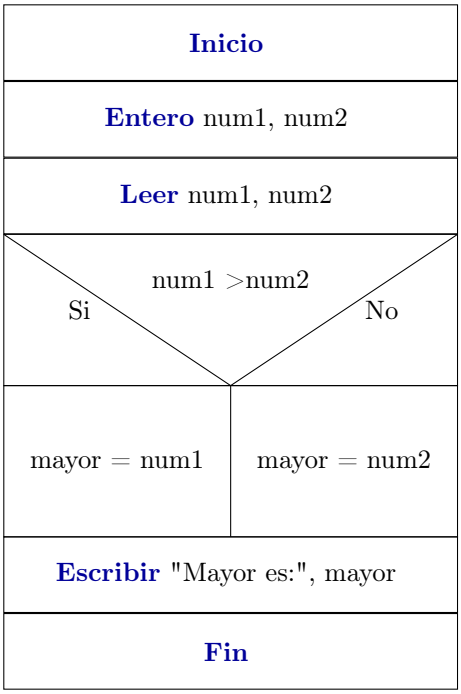


**Ejemplo 4** Dados dos números enteros y diferentes, determinar el mayor de ellos.

**Definición del problema**
Identificar el mayor de dos números

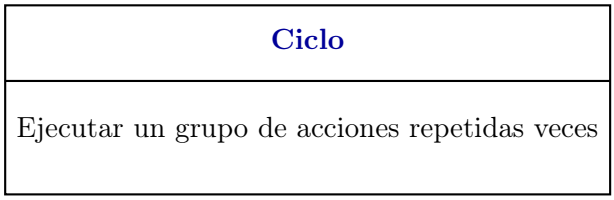
**Análisis del problema**
Datos de entrada: num1,num2
Datos de salida: mayor
Proceso: comparación

**Diseño de la solución**




4. Estucturas de repetición (ciclos o bucles). Las estructuras de repetición (ciclos o bucles) como su nombre lo indica son usadas en la solución de problemas donde es necesario ejecutar repetidas veces una instrucción o un conjunto de instrucciones.

   El número de repeticiones en algunos casos se conoce con anterioridad y en otros no.



(a) Ciclo mientras

**Mientras** condicion **Hacer**

*Instrucciones que se repiten*

**Fin mientras**

- El ciclo mientras comienza evaluando la expresión condición.
- Si el resultado de la evaluación anterior es verdadero, se ejecutarán las instrucciones que estén entre el inicio y el fin del ciclo mientras.
- El ciclo termina cuando la condición deja de cumplirse.

**Ejemplo 5** Escribir un algoritmo con el ciclo **Mientras**, para imprimir los números enteros del 1 al 20.

**Definición del problema**
Imprimir los números del 1 al 20

**Análisis del problema**
Datos de entrada: ninguno
Datos de salida: números enteros del 1 al 20
Proceso: ninguno

**Diseño de la solución**

**Inicio**
**Entero** i = 1

**Mientras** i $\leq$ 20 **Hacer**

**Escribir** i
*i = i + 1*

**Fin mientras**

**Fin**

(b) Con el ciclo **Para** es posible ejecutar una instrucción o un grupo de instrucciones.

Este ciclo maneja el valor inicial, el valor de incremento o decremento y el valor final de variable de control. El ciclo **Para** finaliza cuando la variable de control o contador sobrepasa el valor final.

**Para** variable=inicio **Hasta** fin **Hacer**

*Instrucciones que se repiten*

**Fin para**

**Ejemplo 6** Escribir un algoritmo con el ciclo **Para**, para imprimir los números enteros del 1 al 20.

**Definición del problema**

Imprimir los números del 1 al 20

**Análisis del problema**

Datos de entrada: ninguno
Datos de salida: números enteros del 1 al 20
Proceso: ninguno

**Diseño de la solución**

**Inicio**
**Entero** contador

**Para** contador = 1 **Hasta** 20 **Hacer**

**Escribir** contador

**Fin para**

**Fin**

## 1.3 Ejercicios

### 1.3.1 Estructura Secuencial

1. Diseñar un diagrama Nassi-Schneiderman que lea la base y la altura de un rectángulo, y luego calcule y muestre el área del mismo.

### 1.3.2 Ejercicios con estructura de decisión (Si)

1. Diseñar un diagrama que lea un número y determine si es positivo, negativo o cero, mostrando un mensaje apropiado.

2. Diseñar un diagrama que lea un número entero y determine si es par o impar, mostrando el resultado correspondiente.

3. Diseñar un diagrama que lea la edad de una persona y determine si es mayor de edad (18 años o más), mostrando un mensaje adecuado.

### 1.3.3 Ejercicios con estructura repetitiva (Mientras)

1. Diseñar un diagrama que sume una serie de números ingresados por el usuario hasta que el usuario ingrese un número negativo, momento en el cual se detendrá y mostrará la suma total.

2. Diseñar un diagrama que lea números enteros ingresados por el usuario y cuente cuántos de ellos son mayores que 100. El proceso se detiene cuando se ingrese un número negativo.

3. Diseñar un diagrama que calcule la potencia de un número base elevado a un exponente positivo, usando una estructura mientras.

### 1.3.4 Ejercicios con estructura repetitiva (Para)

1. Diseñar un diagrama que calcule el factorial de un número entero positivo ingresado por el usuario, utilizando una estructura para.

2. Diseñar un diagrama que muestre los primeros 10 números naturales (del 1 al 10), utilizando una estructura para.

3. Diseñar un diagrama que sume todos los números pares comprendidos entre 1 y 50, utilizando una estructura para, y luego muestre el resultado.

# 2

# Iniciando con el lenguaje python

## 2.1 Introducción

Python es uno de los lenguajes de programación más populares y versátiles en la actualidad. Su diseño simple y claro lo hace ideal tanto para principiantes como para desarrolladores experimentados. Con una sintaxis que se asemeja al lenguaje natural, Python reduce la complejidad de escribir código y permite concentrarse en la resolución de problemas en lugar de preocuparse por detalles técnicos.

Este lenguaje es utilizado en una amplia variedad de campos, incluyendo desarrollo web, análisis de datos, inteligencia artificial, automatización, y más. Su comunidad activa y el amplio ecosistema de bibliotecas lo convierten en una herramienta poderosa para resolver problemas de cualquier escala.

El propósito de este capítulo es brindar una introducción práctica a Python y establecer los fundamentos básicos de la programación en este lenguaje. A través de explicaciones claras y ejemplos prácticos, aprenderás cómo trabajar con variables, tipos de datos, estructuras básicas y las operaciones esenciales que necesitarás en proyectos futuros.

## 2.2 Definición de variables, asignación de valores, tipos de datos integrados y conversión de tipos de datos

En cualquier programa, las variables juegan un papel fundamental al permitirnos almacenar y manipular información. En esta sección aprenderás cómo definir variables, asignarles valores, trabajar con los tipos de datos integrados en Python y convertir entre distintos tipos de datos cuando sea necesario.

1. **Definición de variables y asignación de valores**

   Una variable es un contenedor que guarda un valor. En Python, definir una variable es tan sencillo como darle un nombre y asignarle un valor utilizando el operador de asignación (=).

```python
nombre = "Pedro"      # Una variable tipo str (cadena de texto)
edad = 25             # Una variable tipo int (entero)
altura = 1.75         # Una variable tipo float (número decimal)
es_estudiante = True # Una variable tipo bool (booleano)
```

Reglas para nombrar variables en Python: deben comenzar con una letra (a-z, A-Z) o un guion bajo. No pueden empezar con un número. Solo pueden contener letras, números y guión bajo.

Python es sensible a mayúsculas y minúsculas, por lo que edad y Edad son variables diferentes.

2. **Tipos de datos integrados**

Python proporciona varios tipos de datos básicos que cubren la mayoría de las necesidades de programación iniciales. Entre los más comunes están:

Enteros (int): Números enteros positivos o negativos. Ejemplo: -10, 0, 42

Flotantes (float): Números con decimales. Ejemplo: 3.14, -0.001, 2.718

Cadenas de texto (str): Texto encerrado entre comillas simples o dobles. Ejemplo: 'Hola', "Mundo"

Booleanos (bool): Representan valores de verdad: True o False.
Ejemplo: esestudiante = True

Tipos Compuestos (se abordarán en detalle más adelante): Listas (list) Tuplas (tuple) Diccionarios (dict) Conjuntos (set)

Puedes usar la función type() para verificar el tipo de cualquier variable.

```python
x = 42
print(type(x))  # Salida: <class 'int'>
```

3. **Conversión de tipos de datos**

En ocasiones, necesitarás convertir un tipo de dato en otro para realizar ciertas operaciones. Python ofrece funciones integradas para realizar estas conversiones.

(a) **Convertir a entero (int)**

```python
numero_decimal = 3.99
numero_entero = int(numero_decimal)  # Resultado: 3
```

(b) **Convertir a flotante (float)**

```
numero_entero = 42
numero_decimal = float(numero_entero)  # Resultado: 42.0
```

(c) **Convertir a cadena (str)**

```
edad = 25
edad_como_texto = str(edad)  # Resultado: "25"
```

(d) **Convertir a booleano (bool)**

```
texto = ""
es_verdadero = bool(texto)  # Resultado: False (cadena vacía)
```

Nota: No todas las conversiones son posibles. Intentar convertir una cadena no numérica a un entero producirá un error:

```
texto = "Hola"
numero = int(texto)  # Esto generará un ValueError
---------------------------------------------------------------------
ValueError                    Traceback (most recent call last)
<ipython-input-19-1eb1a7569235> in <cell line: 2>()
      1 texto = "Hola"
----> 2 numero = int(texto)  # Esto generará un ValueError
ValueError: invalid literal for int() with base 10: 'Hola'
```

## 2.3 Python como calculadora

Python como calculadora Python no solo es un lenguaje de programación, sino también una poderosa calculadora que permite realizar operaciones matemáticas de forma rápida y sencilla. En esta sección, exploraremos cómo realizar cálculos básicos y avanzados utilizando Python como si fuese una calculadora científica.

1. **Operaciones básicas : Python soporta las operaciones matemáticas más comunes**

   Suma (+)

```
resultado = 5 + 3  # Resultado: 8
print(resultado)
```

   Resta (-)

```
resultado = 10 - 4  # Resultado: 6
print(resultado)
```

Multiplicación (*)

```
resultado = 7 * 6  # Resultado: 42
print(resultado)
```

División (/)

```
resultado = 15 / 3  # Resultado: 5.0 (devuelve número flotante)
print(resultado)
```

Potencia (**)

```
resultado = 2 ** 3  # Resultado: 8 (2 elevado a la 3)
print(resultado)
```

División Entera (//)

```
resultado = 17 // 3  # Resultado: 5 (parte entera del cociente)
print(resultado)
```

Módulo o Resto

```
resultado = 17 % 3  # Resultado: 2 (resto de la división)
print(resultado)
```

2. **Prioridad de operadores**

Python sigue las reglas estándar de precedencia de operadores, también conocidas como PEMDAS: Paréntesis Exponentes Multiplicación y División Adición y Sustracción

```
resultado = 5 + 2 * 3  # Resultado: 11 (multiplica, luego se suma)
print(resultado)
resultado = (5 + 2) * 3  # Resultado: 21 (paréntesis prioridad)
print(resultado)
```

3. **Uso de módulos matemáticos**

Python incluye el módulo math, que proporciona funciones matemáticas avanzadas. Importar el módulo

```
import math
```

Raíz cuadrada (math.sqrt)

```
raiz = math.sqrt(16)  # Resultado: 4.0
print(raiz)
```

Logaritmo natural (math.log)

```
log = math.log(10)  # Logaritmo natural de 10
print(log)
2.302585092994046
```

Funciones trigonométricas

```
seno = math.sin(math.pi / 2)  # Seno de 90 grados
print(seno)
```

## 4. Aplicaciones prácticas

Cálculo del área de un círculo

```
import math
radio = 5
area = math.pi * radio**2
print(f"El área del círculo es: {area}")
El área del círculo es: 78.53981633974483
```

Conversión de grados a radianes

```
grados = 45
radianes = math.radians(grados)
print(f"{grados} grados son {radianes} radianes")
```

Cálculo de interés compuesto

```
capital = 1000  # Capital inicial
tasa = 0.05     # Tasa de interés anual
tiempo = 3      # Años
monto_final = capital * (1 + tasa) ** tiempo
print(f"Monto final: {monto_final}")
Monto final: 1157.6250000000002
```

Conclusión : Usar Python como calculadora no solo simplifica los cálculos cotidianos, sino que también te prepara para trabajar con problemas matemáticos más complejos en tus programas. La flexibilidad y precisión de Python te permitirá explorar aplicaciones en matemáticas, física, economía y más.

## 2.4 Cadenas de caracteres

Las cadenas de caracteres (strings) son uno de los tipos de datos más utilizados en Python. Una cadena es una secuencia de caracteres encerrada entre comillas simples (') o dobles ("). Este tipo de dato se usa para representar texto y permite realizar diversas operaciones útiles.

1. **Creación de cadenas**

   Puedes definir una cadena usando comillas simples o dobles:

```
cadena1 = 'Hola'
cadena2 = "Mundo"
print(cadena1, cadena2)  # Salida: Hola Mundo
```

   También puedes usar triples comillas (''' o """) para cadenas multilínea:

```
mensaje = """Este es un mensaje
que abarca varias líneas."""
print(mensaje)
```

2. **Operaciones básicas**

   Concatenación (+): Une dos cadenas:

```
saludo = "Hola" + " " + "Mundo"
print(saludo)  # Salida: Hola Mundo
```

   Repetición (*): Repite una cadena un número de veces:

```
eco = "Eco " * 3
print(eco)  # Salida: Eco Eco Eco
```

   Longitud de una cadena (len): Devuelve el número de caracteres:

```
longitud = len("Python")
print(longitud)  # Salida: 6
```

3. **Acceso a caracteres**

   Cada carácter de una cadena tiene una posición (índice), empezando desde 0:

```
texto = "Python"
print(texto[0])  # Salida: P (primer carácter)
print(texto[-1]) # Salida: n (último carácter)
```

Nota: Usar índices fuera del rango generará un error:

```
    print(texto[10])  # Genera IndexError
```

4. **Rebanadas (Slicing)**

Puedes extraer partes de una cadena utilizando la sintaxis [inicio:fin:paso]:

```
texto = "Aprender Python"
print(texto[0:8])    # Salida: Aprender
print(texto[9:])     # Salida: Python
print(texto[::-1])   # Salida: nohtyP rednerpA (cadena invertida)
```

5. **Métodos comunes**

Python ofrece numerosos métodos para manipular cadenas. Algunos de los más útiles son:

upper() y lower(): Convierte la cadena a mayúsculas o minúsculas:

```
texto = "Hola Mundo"
print(texto.upper())  # Salida: HOLA MUNDO
print(texto.lower())  # Salida: hola mundo
```

strip(): Elimina espacios en blanco al inicio y al final:

```
texto = "   Hola Mundo   "
print(texto.strip())  # Salida: Hola Mundo
```

replace(): Reemplaza una subcadena por otra:

```
texto = "Python es genial"
print(texto.replace("genial", "fácil"))  # Salida: Python es fácil
```

split() y join(): split() divide la cadena en partes usando un separador:

```
texto = "Uno, Dos, Tres"
partes = texto.split(", ")
print(partes)  # Salida: ['Uno', 'Dos', 'Tres']
```

join() une elementos de una lista en una cadena:

```
palabras = ["Uno", "Dos", "Tres"]
texto = ", ".join(palabras)
print(texto)  # Salida: Uno, Dos, Tres
```

find() y count(): find() devuelve la posición de la primera aparición de una subcadena

```
texto = "Python es divertido"
print(texto.find("divertido"))  # Salida: 10
```

count() cuenta cuántas veces aparece una subcadena:

```
texto = "banana"
print(texto.count("a"))  # Salida: 3
```

6. **Formateo de cadenas**

Python permite insertar valores en cadenas usando varios métodos:

f-strings (recomendado):

```
nombre = "Alice"
edad = 25
print(f"Me llamo {nombre} y tengo {edad} años.")  # Salida: Me llamo Alice y te
```

format():

```
print("Me llamo {} y tengo {} años.".format("Alice", 25))
```

Porcentajes (

```
print("Me llamo %s y tengo %d años." % ("Alice", 25))
```

7. **Aplicaciones prácticas**

Verificar si una palabra está en un texto:

```
texto = "El lenguaje Python es poderoso."
print("Python" in texto)  # Salida: True
```

Inversión de palabras en una oración:

```
texto = "Hola Mundo"
palabras_invertidas = " ".join(texto.split()[::-1])
print(palabras_invertidas)  # Salida: Mundo Hola
```

Contar vocales en una cadena:

```
texto = "Aprender Python"
vocales = sum(1 for letra in texto.lower() if letra in "aeiou")
print(f"El texto tiene {vocales} vocales.")  # Salida: El texto tiene 4 vocales.
```

## 2.5 Listas y Tuplas

Las listas y las tuplas son tipos de datos fundamentales en Python que permiten almacenar colecciones de elementos. Ambos son similares, pero tienen diferencias clave: las listas son mutables (pueden cambiarse), mientras que las tuplas son inmutables (no pueden modificarse después de su creación).

1. **Listas : Definición de una lista**

   Una lista se define usando corchetes ([]) y puede contener elementos de cualquier tipo:

```
mi_lista = [1, 2, 3, "Python", True]
print(mi_lista)  # Salida: [1, 2, 3, 'Python', True]
```

   Las listas también pueden estar vacías:

```
lista_vacia = []
```

2. **Listas : Operaciones básicas**

   Usa índices para acceder a los elementos. Los índices empiezan en 0:

```
numeros = [10, 20, 30, 40]
print(numeros[0])  # Salida: 10
print(numeros[-1]) # Salida: 40
```

   Modificar elementos:

```
numeros[1] = 25
print(numeros)  # Salida: [10, 25, 30, 40]
```

   Agregar elementos: append(): Añade un elemento al final

```
numeros.append(50)
print(numeros)  # Salida: [10, 25, 30, 40, 50]
```

   extend(): Añade varios elementos:

```
numeros.extend([60, 70])
print(numeros)  # Salida: [10, 25, 30, 40, 50, 60, 70]
```

extend(): Eliminar elementos:remove(): Elimina un elemento por valor

```
numeros.remove(25)
print(numeros)  # Salida: [10, 30, 40, 50, 60, 70]
```

pop(): Elimina un elemento por índice (por defecto, el último)

```
numeros.pop()
print(numeros)  # Salida: [10, 30, 40, 50, 60]
```

Longitud de la lista:

```
print(len(numeros))  # Salida: 5
```

3. **Tuplas : Definición de una tupla**

Una tupla se define usando paréntesis (()) o sin ellos:

```
mi_tupla = (1, 2, 3, "Python", True)
print(mi_tupla)  # Salida: (1, 2, 3, 'Python', True)
```

También puedes definir una tupla vacía o de un solo elemento:

```
tupla_vacia = ()
tupla_uno = (5,)  # Importante la coma para que sea tupla
```

4. **Tuplas : Operaciones básicas**

Acceso a elementos: Igual que con listas, mediante índices:

```
numeros = (10, 20, 30, 40)
print(numeros[0])  # Salida: 10
print(numeros[-1]) # Salida: 40
```

Longitud de la tupla:

```
print(len(numeros))  # Salida: 4
```

Desempaquetado: Asignar elementos de la tupla a variables:

```
a, b, c, d = numeros
print(a, b, c, d)  # Salida: 10 20 30 40
```

Convertir entre listas y tuplas:

```
lista = list(numeros)
print(lista)  # Salida: [10, 20, 30, 40]
nueva_tupla = tuple(lista)
print(nueva_tupla)  # Salida: (10, 20, 30, 40)
```

5. **Comparación entre listas y tuplas**

| Característica | Lista | Tupla |
|---|---|---|
| Mutabilidad | Mutable (puede cambiarse) | Inmutable (no puede cambiarse) |
| Definición | Corchetes ([]) | Paréntesis (()) |
| Uso común | Datos que cambian con frecuencia | Datos constantes o que no deben modificarse |

6. **Métodos útiles**

**Métodos para listas**:

sort(): Ordena la lista en su lugar:

```
numeros = [40, 10, 30, 20]
numeros.sort()
print(numeros)  # Salida: [10, 20, 30, 40]
```

index(): Encuentra la posición de un elemento:

```
print(numeros.index(30))  # Salida: 2
```

count(): Cuenta cuántas veces aparece un elemento:

```
print(numeros.count(10))  # Salida: 1
```

**Métodos para tuplas:**

Aunque las tuplas tienen menos métodos, se pueden usar index() y count() igual que en listas:

```
tupla = (10, 20, 30, 20)
print(tupla.count(20))  # Salida: 2
print(tupla.index(30))  # Salida: 2
```

7. **Aplicaciones prácticas**

Almacenar y ordenar una lista de números:

```
numeros = [3, 1, 4, 1, 5, 9]
numeros.sort()
print(numeros)  # Salida: [1, 1, 3, 4, 5, 9]
```

Crear una tupla de coordenadas:

```
coordenada = (10.5, -5.2)
print(f"Latitud: {coordenada[0]}, Longitud: {coordenada[1]}")
```

Convertir una lista de palabras en una tupla inmutable:

```
palabras = ["uno", "dos", "tres"]
palabras_inmutables = tuple(palabras)
print(palabras_inmutables)  # Salida: ('uno', 'dos', 'tres')
```

Conclusión : Listas y tuplas son estructuras esenciales en Python. Las listas son ideales para datos que cambian, mientras que las tuplas son perfectas para datos que deben mantenerse constantes. Dominar su uso te permitirá manejar colecciones de datos de manera eficiente.

## 2.6 Concepto de Rebanadas (Slicing)

El concepto de rebanadas o slicing en Python permite extraer una parte específica de una secuencia, como cadenas de caracteres, listas o tuplas. Slicing utiliza la notación de corchetes con la sintaxis:

```
secuencia[inicio:fin:paso]
```

inicio: Índice del primer elemento que se incluirá. fin: Índice del primer elemento que no se incluirá. paso: Indica cuántos elementos se deben saltar (opcional).

1. **Rebanadas en cadenas:**

   Extraer subcadenas:
   Puedes extraer partes de una cadena mediante slicing

```
texto = "Programación en Python"
subcadena = texto[0:12]
print(subcadena)  # Salida: Programación
```

Omitiendo inicio o fin Si no se especifica inicio, comienza desde el principio. Si no se especifica fin, se incluye hasta el final.

```
print(texto[:11])  # Salida: Programación
print(texto[16:]) # Salida: Python
```

Uso de paso:
Puedes usar paso para omitir elementos o invertir la secuencia:

```
print(texto[0:18:2])  # Salida: Pormcó e
print(texto[::-1])    # Salida: nohtyP ne nóicamargorP
```

2. **Rebanadas en listas**

Seleccionar subconjuntos:
El slicing también funciona con listas para extraer subconjuntos

```
numeros = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
print(numeros[2:7])  # Salida: [2, 3, 4, 5, 6]
```

Omitiendo índices:
Al igual que con cadenas, puedes omitir inicio o fin:

```
print(numeros[:5])  # Salida: [0, 1, 2, 3, 4]
print(numeros[5:])  # Salida: [5, 6, 7, 8, 9]
```

Usando paso:
Puedes especificar un salto en los elementos o invertir la lista:

```
print(numeros[::2])   # Salida: [0, 2, 4, 6, 8]
print(numeros[::-1])  # Salida: [9, 8, 7, 6, 5, 4, 3, 2, 1, 0]
```

3. **Rebanadas en tuplas**

Las tuplas son inmutables, pero también admiten slicing

```
mi_tupla = (10, 20, 30, 40, 50)
print(mi_tupla[1:4])  # Salida: (20, 30, 40)
print(mi_tupla[::-1]) # Salida: (50, 40, 30, 20, 10)
```

```
palabra = "Python"
print(palabra[::-1])  # Salida: nohtyP
```

4. **Ejemplos prácticos**

   (a) Invertir una cadena:

```
        palabra = "Python"
        print(palabra[::-1])  # Salida: nohtyP
```

(b) Seleccionar elementos pares en una lista:

```
numeros = list(range(1, 21))
pares = numeros[1::2]
print(pares)  # Salida: [2, 4, 6, 8, 10, 12, 14, 16, 18, 20]
```

(c) Extraer caracteres alternos de una frase:

```
frase = "Aprender Python es divertido"
print(frase[::2])  # Salida: AeedrPto sditd
```

(d) Seleccionar los últimos 3 elementos de una lista:

```
dias = ["lunes", "martes", "miércoles", "jueves", "viernes"]
print(dias[-3:])  # Salida: ['miércoles', 'jueves', 'viernes']
```

(e) Eliminar elementos de una lista creando una nueva sublista:

```
numeros = [1, 2, 3, 4, 5, 6, 7]
print(numeros[1:-1])  # Salida: [2, 3, 4, 5, 6]
```

5. **Conclusión:**

   El concepto de slicing es una herramienta poderosa en Python para trabajar con secuencias. Permite extraer, reorganizar y manipular datos de forma flexible y eficiente, optimizando muchas operaciones cotidianas en programación.

## 2.7 Condicionales

El condicional if es una estructura fundamental en Python que permite tomar decisiones en el código. Con if, puedes ejecutar diferentes bloques de código dependiendo de si una condición es verdadera o falsa.
**Sintaxis básica:**

```
if condición:
    # Bloque de código si la condición es verdadera
```

Opcionalmente, puedes agregar más casos con elif (else if) y un bloque para manejar el caso contrario con else:

```
if condición:
    # Código si la condición es verdadera
elif otra_condición:
    # Código si otra condición es verdadera
else:
    # Código si ninguna condición es verdadera
```

**Ejemplos:**

1. Verificar si un número es positivo, negativo o cero

```
        numero = int(input("Ingresa un número: "))

        if numero > 0:
            print("El número es positivo.")
        elif numero < 0:
            print("El número es negativo.")
        else:
            print("El número es cero.")
```

2. Calcular el descuento según la cantidad de artículos comprados

```
        cantidad = int(input("¿Cuántos artículos compraste? "))

        if cantidad >= 10:
            descuento = 0.2  # 20% de descuento
        elif cantidad >= 5:
            descuento = 0.1  # 10% de descuento
        else:
            descuento = 0  # Sin descuento

        print(f"Tu descuento es del {descuento * 100}%.")
```

3. Identificar si un año es bisiesto

```
        año = int(input("Ingresa un año: "))

        if (año % 4 == 0 and año % 100 != 0) or (año % 400 == 0):
            print(f"{año} es un año bisiesto.")
        else:
            print(f"{año} no es un año bisiesto.")
```

4. Determinar si una persona es mayor de edad

5. Identificar si un año es bisiesto

```
        edad = int(input("Ingresa tu edad: "))

        if edad >= 18:
            print("Eres mayor de edad.")
        else:
            print("Eres menor de edad.")
```

6. Verificar si un número está en un rango específico

```
        numero = int(input("Ingresa un número: "))

        if 10 <= numero <= 20:
            print(f"El número {numero} está entre 10 y 20.")
        else:
            print(f"El número {numero} no está entre 10 y 20.")
```

## 2.8 Ciclos en Python

Los ciclos son estructuras de control que permiten repetir un bloque de código varias veces. Python ofrece dos tipos principales de ciclos: for y while.

1. **Ciclo for:**

   Se utiliza para iterar sobre una secuencia (como listas, cadenas, o rangos de números). **Sintaxis básica:**

```
        for variable in secuencia:
            # Código que se ejecutará en cada iteración
```

2. **Ciclo while:**

   Repite un bloque de código mientras una condición sea verdadera. **Sintaxis básica:**

```
        while condición:
            # Código que se ejecutará mientras la condición sea verdadera
```

**Ejemplos:**

1. Imprimir números del 1 al 5 con for

```
    for i in range(1, 6):
        print(i)
```

2. Calcular la suma de los números del 1 al 10 con for

```
    suma = 0
    for i in range(1, 11):
        suma += i
    print(f"La suma es: {suma}")
```

3. Iterar sobre una lista de nombres

```
    nombres = ["Ana", "Luis", "María", "Carlos"]
    for nombre in nombres:
        print(f"Hola, {nombre}!")
```

4. Imprimir los caracteres de una cadena

```
    cadena = "Python"
    for caracter in cadena:
        print(caracter)
```

5. Imprimir números pares del 1 al 20

```
    for i in range(2, 21, 2):
        print(i)
```

6. Buscar un elemento en una lista con for y break

```
    numeros = [3, 7, 9, 12, 15]
    buscado = 9

    for numero in numeros:
        if numero == buscado:
            print(f"Encontrado: {buscado}")
            break
    else:
        print("Número no encontrado.")
```

7. Ciclo while para imprimir números del 1 al 5

```
    i = 1
    while i <= 5:
        print(i)
        i += 1
```

8. Calcular el factorial de un número con while

```
    numero = 5
    factorial = 1

    while numero > 0:
        factorial *= numero
        numero -= 1

    print(f"El factorial es: {factorial}")
```

9. Sumar números hasta que el usuario ingrese un número negativo

```
suma = 0

while True:
    numero = int(input("Ingresa un número (negativo para salir): "))
    if numero < 0:
        break
    suma += numero

print(f"La suma total es: {suma}")
```

10. Encontrar el primer múltiplo de 7 mayor que 50

```
numero = 51

while numero % 7 != 0:
    numero += 1

print(f"El primer múltiplo de 7 mayor que 50 es: {numero}")
```

## 2.9 Conjuntos (sets)

Los conjuntos en Python son colecciones desordenadas de elementos únicos. Se utilizan para realizar operaciones matemáticas como unión, intersección y diferencia, y para eliminar duplicados de una colección.

Un conjunto se define utilizando llaves  o la función set().

1. **Creación de conjuntos**

   - **Conjuntos vacíos:** Para crear un conjunto vacío, usa set() (no , ya que esto crea un diccionario):

```
conjunto_vacio = set()
print(conjunto_vacio) # Salida: set()
```

   - **Conjuntos con elementos:** Puedes definir un conjunto directamente con elementos:

```
numeros = {1, 2, 3, 4, 5}
print(numeros)  # Salida: {1, 2, 3, 4, 5}
```

- **Crear conjuntos desde listas o tuplas:** Convierte listas o tuplas en conjuntos para eliminar duplicados:

```
lista = [1, 2, 2, 3, 4, 4]
conjunto = set(lista)
print(conjunto)  # Salida: {1, 2, 3, 4}
```

2. **Operaciones básicas:**

- **Agregar elementos:**
  Usa add() para agregar un elemento:

```
frutas = {"manzana", "naranja"}
frutas.add("plátano")
print(frutas)  # Salida: {'naranja', 'manzana', 'plátano'}
```

- **Eliminar elementos:**
  Usa remove() o discard() para eliminar un elemento:

```
frutas.remove("naranja")
print(frutas)  # Salida: {'manzana', 'plátano'}
```

  La diferencia es que remove() lanza un error si el elemento no existe, mientras que discard() no.

- **Comprobar pertenencia:**
  Verifica si un elemento pertenece al conjunto con el operador in:

```
print("manzana" in frutas)  # Salida: True
```

3. **Operaciones de conjuntos:**
   Python facilita realizar operaciones como unión, intersección y diferencia:

- **Unión:** Combina todos los elementos de dos conjuntos con | o el método union():

```
a = {1, 2, 3}
b = {3, 4, 5}
print(a | b)            # Salida: {1, 2, 3, 4, 5}
print(a.union(b))       # Salida: {1, 2, 3, 4, 5}
```

- **Intersección:** Obtén los elementos comunes entre dos conjuntos con & o intersection():

```
print(a & b)            # Salida: {3}
print(a.intersection(b))  # Salida: {3}
```

- **Diferencia:** Obtén los elementos que están en un conjunto pero no en el otro con - o difference():

```
print(a - b)             # Salida: {1, 2}
print(a.difference(b))   # Salida: {1, 2}
```

- **Diferencia Simétrica:** Obtén los elementos que están en un conjunto o en el otro, pero no en ambos, con ^ o symmetric_difference():

```
print(a ^ b)                    # Salida: {1, 2, 4, 5}
print(a.symmetric_difference(b))  # Salida: {1, 2, 4, 5}
```

4. **Funciones útiles:**
   Algunos métodos importantes de los conjuntos incluyen:

   - **Copiar un conjunto:**

```
nuevo_conjunto = a.copy()
print(nuevo_conjunto)  # Salida: {1, 2, 3}
```

   - **Limpiar un conjunto:**

```
a.clear()
print(a)  # Salida: set()
```

   - **Verificar subconjuntos y superconjuntos:**

```
a = {1, 2}
b = {1, 2, 3}
print(a.issubset(b))  # Salida: True
print(b.issuperset(a))  # Salida: True
```

5. **Ejemplos prácticos**

   - **Eliminar duplicados de una lista:**

```
numeros = [1, 2, 2, 3, 4, 4, 5]
unicos = set(numeros)
print(unicos)  # Salida: {1, 2, 3, 4, 5}
```

   - **Encuentra amigos en común entre dos grupos:**

```
grupo1 = {"Ana", "Luis", "Carlos"}
grupo2 = {"Luis", "Miguel", "Carlos"}
comunes = grupo1 & grupo2
print(comunes)  # Salida: {'Luis', 'Carlos'}
```

- **Crear un conjunto de caracteres únicos en una cadena:**

```
texto = "programación"
caracteres_unicos = set(texto)
print(caracteres_unicos)
# Salida: {'p', 'g', 'm', 'i', 'c', 'ó','a', 'r', 'o', 'n'}
```

- **Verificar si una palabra está en un grupo prohibido:**

```
palabras_prohibidas = {"spam", "publicidad", "fraude"}
mensaje = "Este mensaje contiene spam"
print("spam" in palabras_prohibidas)  # Salida: True
```

- **Calcular los números únicos en varias listas:**

```
lista1 = {1, 2, 3}
lista2 = {3, 4, 5}
numeros_unicos = lista1 | lista2
print(numeros_unicos)  # Salida: {1, 2, 3, 4, 5}
```

## 2.10 Diccionarios

Los diccionarios son estructuras de datos en Python que almacenan pares de clave-valor. Son ideales para organizar información donde cada elemento tiene una clave única que permite su acceso rápido.

Un diccionario se define usando llaves  y cada clave se asocia a un valor mediante el operador :

1. **Creación de diccionarios:**

   - **Diccionario vacío:**
     Puedes crear un diccionario vacío usando  o dict():

```
diccionario_vacio = {}
print(diccionario_vacio)  # Salida: {}
```

   - **Diccionario con claves y valores:**
     Un diccionario puede contener cualquier tipo de clave y valor, siempre que las claves sean inmutables:

```
estudiante = {"nombre": "Ana", "edad": 21, "curso": "Python"}
print(estudiante)
# Salida: {'nombre': 'Ana', 'edad': 21, 'curso': 'Python'}
```

2. **Acceso a elementos:**
   Accede a los valores usando las claves entre corchetes []:

```
print(estudiante["nombre"])  # Salida: Ana
print(estudiante["edad"])    # Salida: 21
```

   Si intentas acceder a una clave que no existe, obtendrás un error. Para evitarlo, usa el método get():

```
print(estudiante.get("apellido", "No disponible"))
# Salida: No disponible
```

3. **Modificar diccionarios:**

   - **Agregar o actualizar valores:**
     Puedes agregar nuevas claves o actualizar los valores existentes:

```
estudiante["apellido"] = "López"  # Agregar clave-valor
estudiante["edad"] = 22           # Actualizar valor
print(estudiante)
# Salida: {'nombre': 'Ana', 'edad': 22, 'curso': 'Python',
# 'apellido': 'López'}
```

   - **Eliminar elementos:**
     Usa del para eliminar un elemento:

```
del estudiante["curso"]
print(estudiante)
# Salida: {'nombre': 'Ana', 'edad': 22, 'apellido': 'López'}
```

     También puedes usar pop() para eliminar y devolver el valor eliminado:

```
apellido = estudiante.pop("apellido")
print(apellido)     # Salida: López
print(estudiante)   # Salida: {'nombre': 'Ana', 'edad': 22}
```

4. **Operaciones comunes:**

   - **Obtener todas las claves:**
     Usa keys() para obtener las claves del diccionario:

```
print(estudiante.keys())
# Salida: dict_keys(['nombre', 'edad'])
```

   - **Obtener todos los valores:**
     Usa values() para obtener los valores:

```
print(estudiante.values())
# Salida: dict_values(['Ana', 22])
```

- **Obtener pares clave-valor:**
  Usa items() para obtener todos los pares:

```
print(estudiante.items())
# Salida: dict_items([('nombre', 'Ana'), ('edad', 22)])
```

5. **Diccionarios anidados:**
   Un diccionario puede contener otros diccionarios como valores:

```
clases = {
    "matemáticas": {"profesor": "Luis", "creditos": 5},
    "historia": {"profesor": "María", "creditos": 3}
          }
print(clases["matemáticas"]["profesor"])  # Salida: Luis
```

6. **Funciones útiles**
   Algunos métodos importantes para trabajar con diccionarios son:

   - **Copiar un diccionario:**

```
copia = estudiante.copy()
print(copia)  # Salida: {'nombre': 'Ana', 'edad': 22}
```

   - **Verificar si una clave existe:**

```
print("nombre" in estudiante)  # Salida: True
print("apellido" in estudiante)  # Salida: False
```

   - **Eliminar todos los elementos:**

```
estudiante.clear()
print(estudiante)  # Salida: {}
```

   - **Actualizar un diccionario:**

```
estudiante.update({"curso": "Python", "nota": 95})
print(estudiante)
# Salida: {'nombre': 'Ana', 'edad': 22, 'curso': 'Python', 'nota': 95}
```

7. **Ejemplos prácticos:**

   - **Contar la frecuencia de elementos en una lista:**

```python
lista = ["rojo", "azul", "rojo", "verde", "azul", "azul"]
frecuencia = {}
for color in lista:
    frecuencia[color] = frecuencia.get(color, 0) + 1
print(frecuencia)
# Salida: {'rojo': 2, 'azul': 3, 'verde': 1}
```

- **Diccionario de precios de productos:**

```python
precios = {"manzana": 1.2, "naranja": 0.8, "plátano": 1.5}
for fruta, precio in precios.items():
    print(f"La {fruta} cuesta ${precio:.2f}")
# Salida:
# La manzana cuesta $1.20
# La naranja cuesta $0.80
# El plátano cuesta $1.50
```

- **Almacenar información de estudiantes:**

```python
estudiantes = {
    "Ana": {"edad": 21, "curso": "Python"},
    "Luis": {"edad": 22, "curso": "Java"}
}
print(estudiantes["Luis"]["curso"])  # Salida: Java
```

- **Conversión de dos listas en un diccionario:**

```python
claves = ["nombre", "edad", "curso"]
valores = ["Ana", 21, "Python"]
estudiante = dict(zip(claves, valores))
print(estudiante)
# Salida: {'nombre': 'Ana', 'edad': 21, 'curso': 'Python'}
```

## 2.11 Usando la comprensión para crear listas:

La comprensión de listas es una forma concisa y elegante de crear listas en Python. Usando una sola línea de código, puedes generar listas basadas en iteraciones o condiciones, lo que las hace más legibles y eficientes.

La sintaxis general de la comprensión de listas es:

```python
[nueva_expresión for elemento in iterable if condición]
```

1. **Crear Listas por comprensión:**

- **Ejemplo 1:** Crear una lista de números
  Generar una lista con los números del 0 al 9:

```
numeros = [x for x in range(10)]
print(numeros)  # Salida: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

- **Ejemplo 2:** Operaciones con los Elementos
  Crear una lista con los cuadrados de los números del 0 al 9:

```
cuadrados = [x**2 for x in range(10)]
print(cuadrados)  # Salida: [0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

2. **Añadir condiciones:**

- **Ejemplo 1:** Filtrar elementos
  Generar una lista con los números pares del 0 al 9:

```
pares = [x for x in range(10) if x % 2 == 0]
print(pares)  # Salida: [0, 2, 4, 6, 8]
```

- **Ejemplo 2:** Operaciones condicionales
  Crear una lista con "par.º ïmpar"según los números del 0 al 9:

```
etiquetas = ["par" if x % 2 == 0 else "impar" for x in range(10)]
print(etiquetas)
# Salida:['par','impar','par','impar','par','impar','par','impar','par','impar']
```

3. **Comprensión con iterables anidados:**

- **Ejemplo 1:** Multiplicar elementos de dos Listas
  Generar una lista con los productos de todos los pares (x, y) donde x está en [1, 2] y y en [3, 4]:

```
productos = [x * y for x in [1, 2] for y in [3, 4]]
print(productos)  # Salida: [3, 4, 6, 8]
```

- **Ejemplo 2:** Matriz bidimensional
  Crear una matriz de 3x3 con comprensión de listas:

```
matriz = [[fila * columna for columna in range(3)] for fila in range(3)]
print(matriz)
# Salida: [[0, 0, 0], [0, 1, 2], [0, 2, 4]]
```

4. **Trabajar con cadenas:**

- **Ejemplo 1:** Convertir letras a mayúsculas
  Crear una lista con las letras de una palabra en mayúsculas:

```
    palabra = "python"
    mayusculas = [letra.upper() for letra in palabra]
    print(mayusculas)  # Salida: ['P', 'Y', 'T', 'H', 'O', 'N']
```

- **Ejemplo 2:** Filtrar letras en una cadena
  Generar una lista con las vocales de una palabra:

```
    palabra = "programación"
    vocales = [letra for letra in palabra if letra in "aeiou"]
    print(vocales)  # Salida: ['o', 'a', 'a', 'i', 'o']
```

5. **Aplicaciones comunes:**

   - **Ejemplo 1:** Crear un diccionario de comprensión de listas
     Generar un diccionario con números y sus cuadrados:

```
    cuadrados = {x: x**2 for x in range(5)}
    print(cuadrados)  # Salida: {0: 0, 1: 1, 2: 4, 3: 9, 4: 16}
```

   - **Ejemplo 2:** Aplanar una lista de listas
     Convertir [[1, 2], [3, 4], [5, 6]] en [1, 2, 3, 4, 5, 6]:

```
    lista_anidada = [[1, 2], [3, 4], [5, 6]]
    plana = [elemento for sublista in lista_anidada for elemento in sublista]
    print(plana)  # Salida: [1, 2, 3, 4, 5, 6]
```

   - **Ejemplo 3:** Filtrar y modificar simultáneamente
     Crear una lista con los números mayores que 5 multiplicados por 2:

```
    numeros = [1, 3, 5, 7, 9]
    resultado = [x * 2 for x in numeros if x > 5]
    print(resultado)  # Salida: [14, 18]
```

## 2.12 Funciones

Las funciones son bloques de código reutilizables que realizan una tarea específica. Son fundamentales para estructurar y organizar programas en Python, ya que permiten dividir un problema en partes más pequeñas y manejables.

1. **Definición de una función:** Para definir una función en Python, se utiliza la palabra clave def, seguida del nombre de la función, paréntesis (que pueden incluir parámetros) y dos puntos. El cuerpo de la función se escribe con sangría.

   - **Ejemplo 1:** Una Función simple

```
    def saludar():
        print("¡Hola, bienvenido a Python!")
    # Llamar a la función
    saludar()
    # Salida: ¡Hola, bienvenido a Python!
```

2. **Parámetros y argumentos:** Las funciones pueden recibir datos (llamados parámetros) para procesarlos.

   - **Ejemplo 1:** Función con parámetros

```
    def saludar_persona(nombre):
        print(f"¡Hola, {nombre}!")

    # Llamar a la función con un argumento
    saludar_persona("Ana")
    # Salida: ¡Hola, Ana!
```

   - **Ejemplo 2:** Función con múltiples parámetros

```
    def sumar(a, b):
        return a + b

    resultado = sumar(5, 3)
    print(resultado)
    # Salida: 8
```

3. **Valores predeterminados:** Los parámetros pueden tener valores predeterminados, que se usan si no se proporciona un argumento al llamar a la función.

   - **Ejemplo 1:** Parámetros con valores predeterminados

```
    def saludar_persona(nombre="invitado"):
        print(f"¡Hola, {nombre}!")

    # Llamar sin argumento
    saludar_persona()
    # Salida: ¡Hola, invitado!

    # Llamar con argumento
    saludar_persona("Luis")
    # Salida: ¡Hola, Luis!
```

4. **Funciones con retorno:** Una función puede devolver un valor utilizando la palabra clave return.

- **Ejemplo 1:** Devolver un valor

```
    def multiplicar(a, b):
        return a * b

    producto = multiplicar(4, 5)
    print(producto)
    # Salida: 20
```

5. **Ámbito de las variables:** Las variables definidas dentro de una función tienen ámbito local, lo que significa que solo existen dentro de la función.

- **Ejemplo 1:** Ámbito local

```
    def ejemplo_ambito():
        x = 10  # Variable local
        print(x)

    ejemplo_ambito()
    # Salida: 10
    # print(x)  # Esto generará un error porque x no existe fuera de la función
```

6. **Funciones lambda:** Las funciones lambda son funciones anónimas (sin nombre) que se definen en una sola línea. Se usan para tareas simples.

- **Ejemplo 1:** Función lambda

```
    cuadrado = lambda x: x**2
    print(cuadrado(5))
    # Salida: 25
```

- **Ejemplo 2:** Usar lambda en funciones integradas

```
    numeros = [1, 2, 3, 4]
    doblados = list(map(lambda x: x * 2, numeros))
    print(doblados)
    # Salida: [2, 4, 6, 8]
```

7. **Funciones recursivas:** Las funciones pueden llamarse a sí mismas. Este mecanismo se conoce como recursión y es útil para resolver problemas como el cálculo de factoriales.

- **Ejemplo 1:** Función recursiva

```
    def factorial(n):
        if n == 0 or n == 1:
            return 1
        else:
```

```
        return n * factorial(n - 1)

print(factorial(5))
# Salida: 120
```

8. **Desempaquetado de argumentos:** Puedes pasar un número variable de argumentos utilizando *args para argumentos posicionales y **kwargs para argumentos con clave.

   - **Ejemplo 11:** Uso de *args

```
def suma(*numeros):
    return sum(numeros)

print(suma(1, 2, 3, 4))
# Salida: 10
```

   - **Ejemplo 2:** Uso de **kwargs

```
def imprimir_info(**datos):
    for clave, valor in datos.items():
        print(f"{clave}: {valor}")

imprimir_info(nombre="Ana", edad=25, ciudad="Madrid")
# Salida:
# nombre: Ana
# edad: 25
# ciudad: Madrid
```

## 2.13 Ejercicios

1. **Introducción a Python**

   - **Ejercicio 1:** Escribe un programa que imprima en la consola el mensaje: "¡Hola! Este es mi primer programa en Python".

2. **Variables y Tipos de Datos**

   - **Ejercicio 2:** Define una variable para almacenar el nombre, la edad, y la ciudad de una persona. Imprime esta información en un formato de oración, por ejemplo: "Hola, me llamo Ana, tengo 25 años y vivo en Madrid."
   - **Ejercicio 3:** Calcula el área de un rectángulo pidiendo al usuario la base y la altura. Usa variables de tipo float.

3. **Python como Calculadora**

- **Ejercicio 4:** Escribe un programa que realice las siguientes operaciones matemáticas en un solo bloque y muestre los resultados: Suma de 25 y 17. Resta de 45 y 19. Multiplicación de 7 y 8. División de 144 entre 12. Potencia de 2 elevado a 5.

4. **Cadenas de Caracteres**

- **Ejercicio 5:** Crea un programa que solicite al usuario su nombre y apellidos, y luego imprima el nombre completo en: Minúsculas. Mayúsculas. Solo con la primera letra de cada palabra en mayúscula.
- **Ejercicio 6:** Escribe un programa que verifique si una palabra ingresada por el usuario es un palíndromo (es decir, se lee igual al derecho y al revés).

5. **Listas y Tuplas**

- **Ejercicio 7:** Crea una lista con las edades de 5 personas. Calcula e imprime:La edad máxima. La edad mínima. El promedio de las edades.
- **Ejercicio 8:** Crea una tupla con los nombres de 5 frutas. Solicita al usuario que escriba el nombre de una fruta y verifica si está en la tupla.

6. **Rebanadas (Slicing)**

- Ejercicio 9: Escribe un programa que: Solicite al usuario que ingrese una palabra. Imprima los tres primeros caracteres, los tres últimos caracteres y la palabra al revés usando slicing.

7. **Condicionales**

- **Ejercicio 10:** Escribe un programa que solicite al usuario un número entero y determine si el número es positivo, negativo o cero. Imprime el resultado correspondiente.
- **Ejercicio 11:** Solicita al usuario dos números enteros y muestra cuál es el mayor, o si ambos son iguales.

8. **Ciclos**

- **Ejercicio 12:** Escribe un programa que imprima la tabla de multiplicar de un número ingresado por el usuario (del 1 al 10) utilizando un ciclo for.
- **Ejercicio 13:** Crea un programa que solicite al usuario que adivine un número secreto (por ejemplo, 7). El programa debe repetirse usando un ciclo while hasta que el usuario adivine correctamente. Al finalizar, imprime un mensaje de felicitación.

9. **Conjuntos**

- **Ejercicio 14:** Crea dos conjuntos de números enteros. Encuentra la intersección de ambos conjuntos. Calcula la unión de ambos conjuntos. Encuentra los elementos que están en el primer conjunto pero no en el segundo.
- **Ejercicio 15:** Escribe un programa que elimine elementos duplicados de una lista usando un conjunto.

10. **Diccionarios**

    - **Ejercicio 16:** Crea un diccionario para almacenar la información de un estudiante (nombre, edad, matrícula, y calificaciones). Luego: Añade una nueva materia con su calificación. Imprime todas las claves y valores del diccionario.
    - **Ejercicio 17:** Escribe un programa que cuente cuántas veces aparece cada palabra en una oración ingresada por el usuario. Usa un diccionario para almacenar los resultados.

11. **Comprensión de Listas**

    - **Ejercicio 18:** Genera una lista con los cuadrados de los números del 1 al 10 utilizando comprensión de listas. **Ejercicio 19:** Crea una lista con las palabras de una oración ingresada por el usuario que tengan más de 5 caracteres.

12. **Funciones**

    - **Ejercicio 20:** Escribe una función que reciba dos números como parámetros y devuelva el mayor de los dos.
    - **Ejercicio 21:** Crea una función que reciba un número como parámetro y determine si es primo.
    - **Ejercicio 22:** Escribe una función que reciba una lista de números y devuelva otra lista con los números pares.
    - **Ejercicio 23:** Implementa una función recursiva que calcule el factorial de un número dado.
    - **Ejercicio 24:** Escribe una función lambda que reciba dos números y devuelva su producto. Luego úsala dentro de una lista de números para calcular los productos con un número constante.

# 3

# Programación orientada a objetos con python

## 3.1 Introducción

La Programación Orientada a Objetos (POO) es un paradigma que ha revolucionado la manera en que desarrollamos software, ofreciendo una metodología robusta y escalable para la creación de aplicaciones. Python, un lenguaje reconocido por su simplicidad y legibilidad, incorpora plenamente este paradigma, permitiendo a los programadores diseñar soluciones más estructuradas, modulares y reutilizables.

Este capítulo explora en detalle cómo Python implementa la POO y cómo puedes aprovechar sus características para construir programas más eficientes y fáciles de mantener. Comenzaremos revisando los conceptos fundamentales de la POO, como clases, objetos, métodos, y atributos. Luego, nos adentraremos en características más avanzadas como la herencia, el polimorfismo, el encapsulamiento y la abstracción, principios clave que permiten manejar la complejidad del software de manera más efectiva.

A lo largo de este capítulo, aprenderás no solo la teoría detrás de la POO, sino también cómo implementarla en Python mediante ejemplos prácticos y casos de uso reales. Descubrirás por qué este enfoque es ampliamente utilizado en la industria, y cómo puede ayudarte a resolver problemas más grandes y complejos de manera más organizada y eficiente.

## 3.2 ¿Qué es la Programación Orientada a Objetos?

La Programación Orientada a Objetos (POO) es un paradigma de programación que organiza el software en torno a objetos, en lugar de acciones o funciones lógicas como en la programación estructurada tradicional. En este enfoque, un programa se ve como una colección de objetos que interactúan entre sí para realizar diversas tareas. Estos objetos son instancias de clases, que definen las características y el comportamiento que los objetos tendrán.

En la POO, un objeto es una representación de una entidad del mundo real o un concepto abstracto que tiene atributos (propiedades o características) y métodos (acciones que puede realizar). Este enfoque permite simular problemas del mundo real de manera más

natural y facilita la creación de programas más complejos y escalables.

Los principales componentes de la POO incluyen:

**Clases:** Son plantillas o modelos a partir de los cuales se crean los objetos. Definen los atributos y los métodos que los objetos asociados tendrán. Objetos: Son instancias de clases. Un objeto toma los atributos y métodos definidos por su clase y puede interactuar con otros objetos en el programa. Atributos: Son las propiedades o datos asociados con un objeto, como el nombre, el color, o cualquier otra característica que lo describa.

**Métodos:** Son las funciones o acciones que un objeto puede realizar. Un método define el comportamiento del objeto, es decir, cómo este interactúa con otros objetos o cómo modifica sus propios atributos. Este enfoque a la programación trae consigo varias ventajas importantes, como la reutilización de código, la modularidad, y la facilidad para gestionar proyectos grandes y complejos. Además, fomenta el desarrollo de soluciones más intuitivas y cercanas a cómo entendemos los problemas en el mundo real.

La POO también se basa en cuatro pilares fundamentales:

1. **Encapsulamiento:** Agrupa datos y funciones dentro de una clase, restringiendo el acceso directo a algunos de los componentes del objeto para proteger su integridad.
2. **Herencia:** Permite crear nuevas clases que extienden el comportamiento de otras, promoviendo la reutilización del código.
3. **Polimorfismo:** Permite que los objetos de diferentes clases puedan ser tratados de manera uniforme, siempre que compartan una interfaz común.
4. **Abstracción:** Facilita el trabajo con conceptos generales sin preocuparse por los detalles de implementación, enfocándose en lo esencial.

## 3.3 Breve historia y contexto

La Programación Orientada a Objetos (POO) surge como una evolución natural de las primeras aproximaciones a la programación estructurada, que dominó las décadas de 1960 y 1970. A medida que los programas comenzaron a crecer en complejidad, surgió la necesidad de desarrollar nuevas formas de organizar el código para hacerlo más comprensible, escalable y fácil de mantener.

El concepto de objetos en la programación tiene sus raíces en los años 60, cuando Ole-Johan Dahl y Kristen Nygaard, en el Centro de Computación Noruego, desarrollaron el lenguaje Simula. Este lenguaje fue diseñado inicialmente para la simulación de sistemas complejos, y fue el primero en introducir el concepto de clases y objetos, conceptos que sentaron las bases de la POO. La capacidad de Simula para modelar entidades del mundo real inspiró a otros a explorar este enfoque.

A mediados de los años 70, el desarrollo de Smalltalk en los laboratorios de Xerox PARC por Alan Kay, Dan Ingalls y otros, consolidó el paradigma de la POO tal como lo conocemos hoy. Smalltalk introdujo la idea de que todo en un programa es un objeto, promoviendo una visión más pura y radical del enfoque orientado a objetos. Este lenguaje influyó en muchos lenguajes de programación modernos, incluido Python.

En las décadas siguientes, la POO fue adoptada gradualmente por la comunidad de desarrolladores, ganando popularidad en los años 80 y 90 con la aparición de lenguajes como C++ y más tarde Java, ambos basados en la POO. Estos lenguajes llevaron el paradigma a una audiencia más amplia, aplicándolo en entornos comerciales e industriales.

Cuando Guido van Rossum creó Python a principios de los 90, su objetivo era diseñar un lenguaje sencillo, legible y fácil de aprender. Aunque inicialmente Python no tenía un fuerte enfoque en la POO, en sus versiones posteriores la POO se integró de manera completa y natural en el lenguaje. Python, gracias a su simplicidad, se convirtió en una excelente opción para aprender y aplicar la POO, facilitando su adopción tanto en aplicaciones académicas como en proyectos industriales.

Hoy en día, la Programación Orientada a Objetos es uno de los paradigmas más utilizados en el mundo del desarrollo de software. Su capacidad para representar de manera clara y estructurada los componentes de un sistema complejo ha hecho que sea ampliamente adoptado en el desarrollo de aplicaciones a gran escala, juegos, inteligencia artificial, simulaciones y más.

En resumen, la POO ha evolucionado a lo largo de varias décadas, pasando de ser un concepto de simulación a convertirse en una piedra angular en el desarrollo de software moderno. Python, con su enfoque en la simplicidad y la legibilidad, ha adoptado este paradigma de manera flexible, lo que lo convierte en un excelente lenguaje para implementar la POO en proyectos tanto pequeños como grandes.

## 3.4 Razones para usar Programación Orientada a Objetos (POO)

La Programación Orientada a Objetos (POO) se ha convertido en uno de los paradigmas de programación más utilizados en el desarrollo de software moderno. Este enfoque ofrece múltiples ventajas que permiten crear aplicaciones más estructuradas, flexibles y fáciles de mantener. A continuación, se describen las principales razones para utilizar POO en el desarrollo de software:

1. **Modularidad y organización del código**

   En POO, los programas se dividen en clases y objetos, lo que facilita la organización del código. Esta estructura modular permite que cada parte del código se enfoque en una tarea específica, haciendo más sencillo localizar, entender y modificar componentes del programa sin afectar el resto del sistema. Esto resulta especialmente útil en proyectos

grandes y complejos, donde es fundamental tener un código bien estructurado y organizado.

2. **Reutilización de código**

   Uno de los grandes beneficios de la POO es la capacidad de reutilizar código a través de la herencia. Las clases pueden compartir atributos y métodos entre sí, lo que permite crear nuevas clases basadas en las ya existentes. Esto evita la duplicación de código y facilita la expansión de funcionalidades en aplicaciones sin tener que reescribir todo desde cero.

3. **Facilita el mantenimiento y la escalabilidad**

   El uso de objetos y clases en la POO facilita la identificación y corrección de errores. Como el código está encapsulado en pequeñas unidades funcionales (objetos), es más fácil aislar y corregir errores sin afectar otras partes del programa. Además, la capacidad de agregar nuevas clases o modificar las existentes sin comprometer la estructura global hace que sea más sencillo escalar aplicaciones a medida que crecen en tamaño y complejidad.

4. **Simulación del mundo real**

   La POO permite modelar conceptos y objetos del mundo real en el software, lo que la hace intuitiva y natural para muchos tipos de problemas. Este paradigma es especialmente útil para aplicaciones en campos como la simulación, el desarrollo de videojuegos o la inteligencia artificial, donde los objetos y sus interacciones pueden reflejar entidades y comportamientos del mundo real.

5. **Mejor manejo de la complejidad**

   A medida que los programas se vuelven más grandes y complejos, la POO proporciona un marco que ayuda a manejar esa complejidad. A través de la abstracción, es posible enfocarse en los aspectos más importantes de un problema, dejando los detalles de implementación en segundo plano. El encapsulamiento ayuda a ocultar el funcionamiento interno de los objetos, exponiendo solo lo necesario al exterior, lo que simplifica la interacción entre distintas partes del programa.

6. **Facilita el trabajo en equipo**

   En proyectos de desarrollo colaborativo, la POO es especialmente útil, ya que permite que los equipos trabajen en diferentes partes de un sistema sin interferir con el trabajo de otros. Cada desarrollador puede enfocarse en una clase o conjunto de clases, lo que promueve un desarrollo más eficiente y coordinado. Las interfaces y la claridad en las relaciones entre objetos permiten que el trabajo en equipo sea más fluido y organizado.

7. **Flexibilidad y extensibilidad**

   La POO facilita la creación de sistemas flexibles que pueden adaptarse y evolucionar

con el tiempo. Gracias al polimorfismo, los objetos de diferentes clases pueden ser tratados de manera uniforme, lo que permite cambiar o ampliar el comportamiento de un sistema sin modificar grandes secciones de código. Además, al utilizar la herencia, es posible extender las funcionalidades de las clases existentes sin alterar su código original.

8. **Facilita el uso de bibliotecas y frameworks**

   Muchos de los frameworks y bibliotecas más populares están diseñados con POO, lo que significa que comprender y utilizar este paradigma permite aprovechar todo el potencial de estas herramientas. Esto es especialmente útil en el desarrollo web, aplicaciones móviles, inteligencia artificial, y otros campos donde el uso de bibliotecas especializadas es común.

## 3.5 Ventajas y Desventajas de la Programación Orientada a Objetos (POO)

La Programación Orientada a Objetos (POO) es ampliamente utilizada en el desarrollo de software moderno debido a los beneficios que aporta en términos de organización y escalabilidad. Sin embargo, como todo paradigma, también presenta algunas desventajas que es importante tener en cuenta. A continuación, se describen las principales ventajas y desventajas de utilizar POO.

### Ventajas de la POO

1. **Modularidad y organización del código**

   La POO facilita la creación de programas bien organizados, donde el código se estructura en clases y objetos. Esto permite dividir el código en pequeñas unidades lógicas que pueden ser desarrolladas y mantenidas de forma independiente. Esta modularidad hace que el software sea más fácil de leer, mantener y depurar.

2. **Reutilización de código**

   Gracias a la herencia, las clases pueden compartir características y comportamiento, lo que permite reutilizar código y extender funcionalidades sin duplicarlo. Esto ahorra tiempo y esfuerzo, especialmente en proyectos grandes, al evitar escribir el mismo código en múltiples lugares.

3. **Escalabilidad**

   La POO permite crear sistemas escalables, ya que es fácil agregar nuevas funcionalidades o modificar las existentes sin afectar el resto del sistema. A medida que las aplicaciones crecen, la estructura orientada a objetos ayuda a manejar la complejidad sin comprometer la claridad o la funcionalidad del código.

4. **Mantenimiento sencillo**

El uso de encapsulamiento protege los datos y permite controlar cómo se accede y modifica el estado interno de los objetos. Esto reduce la posibilidad de errores y hace que los programas sean más fáciles de mantener, ya que los cambios realizados en una parte del código no afectan al resto del sistema si está bien encapsulado.

5. **Simulación del mundo real**

   La POO facilita el modelado de problemas del mundo real, ya que los objetos pueden representar entidades físicas o abstractas. Esto hace que los programas orientados a objetos sean intuitivos y comprensibles, especialmente en áreas como simulaciones, videojuegos o aplicaciones de inteligencia artificial.

6. **Facilita el trabajo en equipo**

   Al dividir el código en clases y objetos, es más fácil que varios desarrolladores trabajen en diferentes partes del mismo proyecto sin interferir entre sí. La POO fomenta la colaboración y la integración de múltiples componentes desarrollados por distintos equipos.

**Desventajas de la POO**

1. **Curva de aprendizaje**

   Para quienes están acostumbrados a paradigmas más simples, como la programación estructurada, aprender POO puede ser un desafío. Los conceptos de clases, objetos, herencia y polimorfismo pueden resultar difíciles de entender al principio, especialmente para los programadores novatos.

2. **Sobrecarga en la gestión del código**

   En algunos casos, el uso de POO puede llevar a una sobreingeniería. El diseño orientado a objetos implica crear clases y objetos para todo, lo que puede complicar programas que podrían ser más sencillos si se utilizaran enfoques más directos o procedimentales. Esto puede aumentar la complejidad innecesariamente para proyectos pequeños o con requisitos simples.

3. **Consumo de memoria**

   Los objetos suelen ocupar más memoria que las estructuras de datos simples. En sistemas donde los recursos de memoria son limitados, como en dispositivos embebidos, el uso intensivo de objetos puede no ser la mejor opción. Además, la creación y destrucción de objetos puede aumentar la sobrecarga en el uso de recursos.

4. **Menor eficiencia en ciertos casos**

   Aunque la POO ofrece muchas ventajas en términos de organización y mantenimiento, puede no ser la opción más eficiente para tareas que requieren alto rendimiento, como el procesamiento intensivo de datos o aplicaciones en tiempo real. Los métodos de

acceso y la creación de objetos pueden introducir una sobrecarga que no se presenta en enfoques más directos, como la programación estructurada.

5. **Dificultad para representar ciertos problemas**

   No todos los problemas se ajustan bien al modelo de objetos. En algunos casos, los datos y las funciones pueden no encajar naturalmente en el marco de objetos y clases, lo que puede hacer que la solución sea más complicada de lo necesario. Para ciertos tipos de algoritmos, la programación funcional o estructurada puede ser más adecuada.

En resumen, la Programación Orientada a Objetos presenta una gran cantidad de ventajas en términos de organización, reutilización de código y facilidad de mantenimiento. Sin embargo, es importante tener en cuenta sus posibles desventajas, especialmente en términos de complejidad y sobrecarga de recursos, para elegir el enfoque más adecuado según el tipo de proyecto.

## 3.6 Objetos, clases, atributos y métodos

En la Programación Orientada a Objetos (POO), los conceptos clave que forman la base de este paradigma son los objetos, las clases, los atributos y los métodos. Estos elementos permiten organizar y estructurar el código de una manera más cercana a cómo representamos los problemas en el mundo real.

1. **Objetos**

   Un objeto es una instancia de una clase y representa una entidad del mundo real o un concepto dentro del programa. Cada objeto tiene un estado (definido por sus atributos) y un comportamiento (definido por sus métodos).

   En términos simples, los objetos son çosas"que interactúan entre sí dentro de un programa. Un objeto puede ser algo físico, como un çoche.º una "persona", o algo abstracto, como una "transacción bancaria". Cada objeto es único, aunque puede compartir características y comportamientos con otros objetos de la misma clase.

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

# Creación de un objeto de la clase Coche
mi_coche = Coche("Toyota", "Corolla")
```

   En este ejemplo, **mi_coche** es un objeto de la clase **Coche**, con los atributos marca y modelo.

2. **Clases**

Una clase es una plantilla o modelo a partir de la cual se crean los objetos. Define las propiedades (atributos) y comportamientos (métodos) que tendrán los objetos basados en esa clase. Las clases permiten agrupar características comunes de varios objetos en un solo lugar, lo que facilita la reutilización del código.

En Python, las clases se definen con la palabra clave `class`, y dentro de ellas se establecen los atributos y métodos que los objetos de esa clase tendrán.

**Ejemplo de clase:**

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

    def arrancar(self):
        print(f"El {self.marca} {self.modelo} ha arrancado.")
```

En este caso, la clase **Coche** tiene dos atributos: marca y modelo, y un método llamado **arrancar**. Los objetos creados a partir de esta clase heredarán estas características.

3. **Atributos**

Los atributos son las propiedades o características que definen el estado de un objeto. Los atributos permiten almacenar información dentro de un objeto. Por ejemplo, en un objeto **Coche**, los atributos podrían ser marca, modelo y color.

En Python, los atributos se definen dentro de un método especial llamado `__init__`, que es el constructor de la clase. El constructor inicializa los atributos cuando se crea un objeto. Los atributos pueden ser modificados después de la creación del objeto si es necesario.

**Ejemplo de atributos:**

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

# Crear un objeto de la clase Coche
mi_coche = Coche("Toyota", "Corolla", "Rojo")

# Acceder a los atributos del objeto
print(mi_coche.marca)  # Toyota
print(mi_coche.color)  # Rojo
```

En este ejemplo, **mi_coche** tiene los atributos marca, modelo y color, que se asignan cuando se crea el objeto.

4. **Métodos**

   Los métodos son funciones que definen el comportamiento de los objetos, es decir, las acciones que pueden realizar. Los métodos son definidos dentro de una clase y son ejecutados por los objetos. Un método puede acceder a los atributos del objeto y modificarlos, o interactuar con otros objetos y realizar operaciones.

   En Python, los métodos se definen dentro de las clases utilizando funciones comunes, y el primer parámetro de cada método es siempre **self**, que hace referencia al propio objeto.

   **Ejemplo de método:**

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

    def arrancar(self):
        print(f"El {self.marca} {self.modelo} ha arrancado.")

# Crear un objeto de la clase Coche
mi_coche = Coche("Toyota", "Corolla")

# Llamar al método arrancar del objeto
mi_coche.arrancar()
```

   En este ejemplo, el método **arrancar** imprime un mensaje que indica que el coche ha arrancado. Este comportamiento es específico del objeto **mi_coche**, pero otros objetos de la misma clase **Coche** también podrían tener el mismo comportamiento.

## 3.7 El método __init__ y sus características

Algunas de sus características:

1. **Inicialización de instancias:** Inicializa las propiedades de un objeto cuando se crea una nueva instancia de una clase. Se llama automáticamente cuando se crea un objeto de la clase.

2. **Método especial:** Támbien conocido como constructor en otros lenguajes de programación. Su nombre tiene dobles guiones bajos antes y después para indicar que es un método especial de python.

3. **No es un constructor:** Aunque comúnmente se llama constructor, técnicamente **__init__** no es un constructor en python, ya que no crea el objeto. El objeto ya ha sido creado por el método **__new__**. **__init__** solo inicializa el objeto.

4. **Parámetro self:** El primer parámetro de **__init__** es siempre **self**, que es una referencia al objeto actual que se está creando. Esto permite que **__init__** acceda y modifique los atributos de la instancia.

5. **Párametros adicionales:** Además de **self**, **__init__** puede aceptar otros parámetros que se utilizan para inicializar los atributos del objeto. Estos parámetros se pasan cuando se crea una nueva instancia de la clase.

6. **No devuelve valor:** A diferencia de algunos otros métodos, **__init__** no devuelve valor alguno. Su propósito es modificar el objeto en sí, no producir una salida separada.

## 3.8 Mensajes

La comunicación entre los objetos recibe el nombre de mensaje.

Cuando se llama un método de un objeto, se dice que se está enviando un mensaje al objeto.

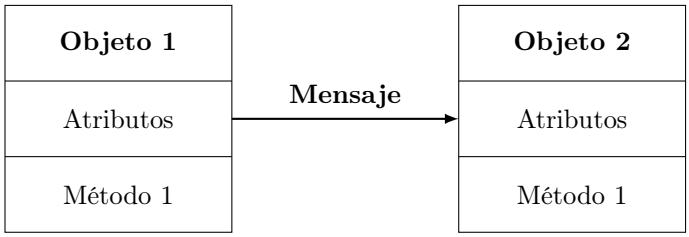



**Figura 3.1:** Comunicación entre Objeto 1 y Objeto 2 mediante un mensaje

**Mensaje:** Estimula la ocurrencia de cierto comportamiento en el receptor. El comportamiento se realiza cuando se ejecuta el método.

Una operación dentro de un objeto emisor genera un mensaje de la forma:

**destino.operación(parámetros)**

Donde:

- **destino :** Objeto receptor el cual es estimulado por el mensaje.
- **operación:** Es el método que recibe el mensaje.
- **parámetros:** Es la información requerida por el mensaje

## 3.9 Ejemplo 1: Sistema de gestión de estudiantes

Descripción: Este programa permite gestionar la información básica de estudiantes. Utiliza una clase llamada **Estudiante** que almacena el nombre, la edad y la nota de un estudiante. Además, permite mostrar su información y determinar si ha aprobado o no.

### Esquema de la clase Estudiante

- **Clase: Estudiante**
  - Atributos:
    - **nombre:** Nombre del estudiante.
    - **edad:** Edad del estudiante.
    - **nota:** Nota del estudiante.
  - Métodos:
    - `__init__(self, nombre, edad, nota)`: Constructor que inicializa los atributos.
    - `mostrar_informacion(self)`: Muestra la información del estudiante.
    - `ha_aprobado(self)`: Verifica si el estudiante ha aprobado (nota mayor o igual a 6).

### Código en python

```python
# Definición de la clase Estudiante
class Estudiante:
    def __init__(self, nombre, edad, nota):
        """Constructor para inicializar los atributos nombre, edad y nota"""
        self.nombre = nombre
        self.edad = edad
        self.nota = nota

    def mostrar_informacion(self):
        """Muestra la información del estudiante"""
        print(f"Nombre: {self.nombre}")
        print(f"Edad: {self.edad}")
        print(f"Nota: {self.nota}")

    def ha_aprobado(self):
        """Determina si el estudiante ha aprobado según su nota"""
        if self.nota >= 6:
            print(f"{self.nombre} ha aprobado.")
        else:
            print(f"{self.nombre} no ha aprobado.")

# Ejemplo de uso del programa
estudiante1 = Estudiante("Juan Pérez", 20, 8.5)
estudiante2 = Estudiante("Ana Gómez", 22, 5.7)

# Mostrar la información y determinar si han aprobado
print("Información del Estudiante 1:")
estudiante1.mostrar_informacion()
estudiante1.ha_aprobado()
```

```
30
31 print("\nInformación del Estudiante 2:")
32 estudiante2.mostrar_informacion()
33 estudiante2.ha_aprobado()
```

**Explicación detallada**

1. **Clase Estudiante:** Define la estructura para crear objetos que representan estudiantes, cada uno con un nombre, edad, y nota.
2. **Método mostrar_informacion:** Imprime los datos del estudiante.
3. **Método ha_aprobado:** Verifica si el estudiante ha aprobado basándose en su nota.
4. **Creación de objetos:** Se crean dos estudiantes con diferentes valores de atributos.
5. **Llamada a métodos:** Se llama a `mostrar_informacion()` para cada estudiante, y luego se determina si ha aprobado o no usando `ha_aprobado()`.

## 3.10 Ejemplo 2: Control de inventario en una tienda

Descripción: Este programa permite gestionar el inventario de una tienda mediante una clase llamada **Producto**. Cada objeto de la clase representará un producto con un nombre, un precio y una cantidad en stock. El programa incluye métodos para mostrar la información del producto y para modificar la cantidad disponible en el inventario (agregar o restar unidades).

**Esquema de la clase Producto**

- **Clase: Producto**
  - Atributos:
    - **nombre:** Nombre del producto.
    - **precio:** Precio del producto.
    - **cantidad:** Cantidad disponible en el inventario.
  - Métodos:
    - `__init__(self, nombre, precio, cantidad)`: Constructor que inicializa los atributos.
    - `mostrar_informacion(self)`: Muestra la información del producto.
    - `agregar_stock(self, cantidad)`: Agrega una cantidad específica de unidades al inventario.
    - `vender_producto(self, cantidad)`: Reduce la cantidad de unidades en el inventario si hay stock suficiente.

**Código en python**

```python
# Definición de la clase Producto
class Producto:
    def __init__(self, nombre, precio, cantidad):
        """Constructor para inicializar los atributos nombre, precio y cantidad"""
        self.nombre = nombre
        self.precio = precio
        self.cantidad = cantidad

    def mostrar_informacion(self):
        """Muestra la información del producto"""
        print(f"Producto: {self.nombre}")
        print(f"Precio: ${self.precio}")
        print(f"Cantidad en stock: {self.cantidad}")

    def agregar_stock(self, cantidad):
        """Agrega una cantidad específica de unidades al inventario"""
        self.cantidad += cantidad
        print(f"Se han agregado {cantidad} unidades. Stock actual: {self.cantidad}")

    def vender_producto(self, cantidad):
        """Reduce la cantidad de unidades en el inventario si hay stock suficiente"""
        if self.cantidad >= cantidad:
            self.cantidad -= cantidad
            print(f"Se han vendido {cantidad} unidades de {self.nombre}. Stock restante:
        else:
            print(f"No hay suficientes unidades de {self.nombre} en stock para completar

# Ejemplo de uso del programa
producto1 = Producto("Laptop", 1000, 10)
producto2 = Producto("Smartphone", 500, 25)

# Mostrar información de los productos
print("Información del Producto 1:")
producto1.mostrar_informacion()

print("\nInformación del Producto 2:")
producto2.mostrar_informacion()

# Agregar stock y vender productos
print("\nOperaciones de inventario:")
producto1.agregar_stock(5)
producto1.vender_producto(3)

producto2.vender_producto(30)
producto2.agregar_stock(10)
```

**Explicación detallada**

1. **Clase Producto:** Representa un producto en el inventario de una tienda con atributos como nombre, precio y cantidad en stock.
2. **Métodos:**
   - `__init__`: Inicializa los atributos del producto.
   - `mostrar_informacion`: Muestra los datos del producto.
   - `agregar_stock`: Aumenta el inventario disponible.
   - `vender_producto`: Disminuye el stock si hay suficiente cantidad disponible.
3. **Creación de objetos:** Se crean dos productos: **Laptop** y **Smartphone**, con diferentes valores de precio y cantidad en stock.
4. **Operaciones de inventario:** Se agrega stock, se venden unidades, y se intenta vender más unidades de las disponibles para mostrar el manejo de errores.

## 3.11 Ejemplo 3: Simulación de una cuenta bancaria

Descripción: Este programa permite simular una cuenta bancaria mediante una clase llamada **CuentaBancaria**. Cada objeto de esta clase representará una cuenta con un titular, un saldo y un tipo de cuenta (ahorros o corriente). El programa incluye métodos para depositar y retirar dinero de la cuenta, así como para mostrar el saldo actual.

**Esquema de la clase CuentaBancaria**

- **Clase: CuentaBancaria**
  - Atributos:
    - **titular:** Nombre del titular de la cuenta.
    - **saldo:** Saldo disponible en la cuenta.
    - **tipo_cuenta:** Tipo de cuenta (puede ser .ªhorros.º çorriente").
  - Métodos:
    - `__init__(self, titular, saldo, tipo_cuenta)`: Constructor que inicializa los atributos.
    - `mostrar_saldo(self)`: Muestra el saldo actual de la cuenta.
    - `depositar(self, cantidad)`: Aumenta el saldo de la cuenta con la cantidad especificada.
    - `retirar(self, cantidad)`: Reduce el saldo de la cuenta si hay suficiente dinero disponible.

**Código en python**

```python
# Definición de la clase CuentaBancaria
class CuentaBancaria:
    def __init__(self, titular, saldo, tipo_cuenta):
        """Constructor para inicializar los atributos titular, saldo y tipo de cuenta"""
        self.titular = titular
        self.saldo = saldo
        self.tipo_cuenta = tipo_cuenta

    def mostrar_saldo(self):
        """Muestra el saldo actual de la cuenta"""
        print(f"Titular: {self.titular}")
        print(f"Tipo de cuenta: {self.tipo_cuenta}")
        print(f"Saldo actual: ${self.saldo}")

    def depositar(self, cantidad):
        """Aumenta el saldo de la cuenta con la cantidad especificada"""
        self.saldo += cantidad
        print(f"Has depositado ${cantidad}. Saldo actual: ${self.saldo}")

    def retirar(self, cantidad):
        """Reduce el saldo de la cuenta si hay suficiente dinero disponible"""
        if self.saldo >= cantidad:
            self.saldo -= cantidad
            print(f"Has retirado ${cantidad}. Saldo restante: ${self.saldo}")
        else:
            print(f"No tienes suficiente saldo para retirar ${cantidad}. Saldo actual: $

# Ejemplo de uso del programa
cuenta1 = CuentaBancaria("Juan Pérez", 1500, "ahorros")
cuenta2 = CuentaBancaria("Ana Gómez", 800, "corriente")

# Mostrar saldo de las cuentas
print("Información de la Cuenta 1:")
cuenta1.mostrar_saldo()

print("\nInformación de la Cuenta 2:")
cuenta2.mostrar_saldo()

# Realizar operaciones en las cuentas
print("\nOperaciones en la Cuenta 1:")
cuenta1.depositar(500)
cuenta1.retirar(1000)

print("\nOperaciones en la Cuenta 2:")
cuenta2.retirar(900)
cuenta2.depositar(200)
```

**Explicación detallada**

1. **Clase CuentaBancaria:** Esta clase representa una cuenta bancaria, que incluye información sobre el titular de la cuenta, el saldo disponible y el tipo de cuenta (puede ser .ªhorros.º çorriente").

2. **Métodos:**

   - `__init__`: El constructor inicializa los atributos de la clase (titular, saldo, y tipo_cuenta) al crear un objeto de la clase.
   - `mostrar_saldo`: Este método imprime en la consola la información del titular, el tipo de cuenta y el saldo actual.
   - `depositar`: Aumenta el saldo de la cuenta en la cantidad especificada.
   - `retirar`: Este método reduce el saldo de la cuenta si hay suficiente dinero disponible. Si no hay suficiente saldo, muestra un mensaje de error.

3. **Creación de objetos:** Se crean dos instancias de la clase CuentaBancaria:

   - **cuenta1**, que pertenece a Juan Pérez, es de tipo .ªhorrosz tiene un saldo inicial de $1500.
   - **cuenta2**, que pertenece a Ana Gómez, es de tipo çorrientez tiene un saldo inicial de $800.

4. **Llamada a los métodos:**

   - Primero, se llama al método `mostrar_saldo` para mostrar la información de ambas cuentas.
   - Luego, se realizan varias operaciones de depósito y retiro:
     - En la cuenta de Juan Pérez, se deposita $500, y luego se retiran $1000.
     - En la cuenta de Ana Gómez, se intenta retirar $900, pero como no hay suficiente saldo, se muestra un mensaje de error. Posteriormente, se deposita $200.

## 3.12 Ejercicios

**Ejercicio 1: Sistema de Gestión de Libros**

Escribe un programa que implemente una clase llamada **Libro**. Cada objeto de la clase deberá tener los atributos `titulo`, `autor` y `precio`. El programa deberá incluir métodos para:

1. Inicializar los atributos del libro.

2. Mostrar la información del libro.

3. Aplicar un descuento al precio del libro.

El programa debe permitir crear varios objetos de la clase **Libro**, mostrar su información y aplicar descuentos.

**Ejercicio 2: Sistema de Control de Empleados**

Desarrolla un programa que implemente una clase llamada **Empleado**. Cada empleado debe tener los atributos `nombre`, `sueldo` y `puesto`. El programa deberá incluir métodos para:

1. Inicializar los atributos del empleado.
2. Mostrar la información del empleado.
3. Calcular el salario mensual de un empleado considerando que se le debe retener un 10 % en impuestos.

El programa debe permitir crear objetos de la clase **Empleado**, mostrar la información y calcular su salario mensual después de impuestos.

**Ejercicio 3: Sistema de Gestión de Productos**

Escribe un programa que implemente una clase llamada **ProductoElectronico**. Cada producto deberá tener los atributos `nombre`, `precio` y `garantia` (en años). El programa deberá incluir métodos para:

1. Inicializar los atributos del producto.
2. Mostrar la información del producto.
3. Verificar si el producto está dentro del periodo de garantía, comparando los años con un límite definido.

El programa debe permitir crear varios objetos y verificar si los productos están en garantía.

**Ejercicio 4: Gestión de una Biblioteca**

Crea un programa que implemente una clase llamada **Biblioteca**. Cada objeto debe tener los atributos `nombre` y `libros` (una lista de libros). El programa debe incluir métodos para:

1. Inicializar la biblioteca con un nombre y una lista de libros.
2. Mostrar la lista de libros disponibles.
3. Agregar un libro a la lista de libros disponibles.

El programa debe permitir gestionar múltiples bibliotecas y agregar libros a ellas.

**Ejercicio 5: Sistema de Reservas de Hotel**

Escribe un programa que implemente una clase llamada **HabitacionHotel**. Cada habitación debe tener los atributos `numero`, `tipo` (individual, doble, suite) y `ocupada` (booleano que indica si está ocupada o no). El programa debe incluir métodos para:

1. Inicializar los atributos de la habitación.
2. Mostrar la información de la habitación.
3. Cambiar el estado de ocupación de la habitación (de ocupada a libre y viceversa).

El programa debe permitir gestionar las habitaciones de un hotel y modificar su estado de ocupación.

**Ejercicio 6: Sistema de Gestión de Vehículos**

Desarrolla un programa que implemente una clase llamada **Vehiculo**. Cada vehículo debe tener los atributos `marca`, `modelo` y `kilometraje`. El programa debe incluir métodos para:

1. Inicializar los atributos del vehículo.
2. Mostrar la información del vehículo.
3. Registrar un nuevo viaje que aumente el kilometraje del vehículo.

El programa debe permitir crear varios objetos de la clase **Vehiculo** y registrar los viajes realizados.

**Ejercicio 7: Sistema de Gestión de Cursos**

Escribe un programa que implemente una clase llamada **Curso**. Cada curso deberá tener los atributos `nombre`, `codigo` y `creditos`. El programa deberá incluir métodos para:

1. Inicializar los atributos del curso.
2. Mostrar la información del curso.
3. Calcular el número de horas del curso, asumiendo que cada crédito equivale a 10 horas de clase.

El programa debe permitir crear varios objetos de la clase **Curso** y calcular las horas de clase correspondientes a cada curso.

**Ejercicio 8: Simulación de una Cuenta de Ahorros**

Desarrolla un programa que implemente una clase llamada **CuentaAhorros**. Cada cuenta debe tener los atributos `titular`, `saldo`, y `interes` (porcentaje de interés anual). El programa deberá incluir métodos para:

1. Inicializar los atributos de la cuenta.
2. Mostrar el saldo de la cuenta.
3. Aplicar el interés anual al saldo, calculando los intereses generados.

El programa debe permitir calcular los intereses de varias cuentas de ahorro.

**Ejercicio 9: Sistema de Control de Inventario**

Crea un programa que implemente una clase llamada **Articulo**. Cada artículo debe tener los atributos `nombre`, `precio` y `stock`. El programa debe incluir métodos para:

1. Inicializar los atributos del artículo.
2. Mostrar la información del artículo.
3. Actualizar el precio del artículo.
4. Realizar una venta que reduzca el stock, si hay suficiente cantidad disponible.

El programa debe permitir gestionar el inventario de artículos y realizar ventas.

**Ejercicio 10: Simulación de un Vehículo Eléctrico**

Escribe un programa que implemente una clase llamada **VehiculoElectrico**. Cada vehículo debe tener los atributos `marca`, `modelo`, `bateria` (capacidad en kWh) y `autonomia` (en kilómetros). El programa debe incluir métodos para:

1. Inicializar los atributos del vehículo.
2. Mostrar la información del vehículo.
3. Calcular la autonomía restante después de un viaje, basado en el consumo de batería por kilómetro.

El programa debe permitir gestionar vehículos eléctricos y calcular la autonomía después de cada viaje.

## 3.13 Herencia, Polimorfismo, Encapsulamiento y Abstracción

Los pilares fundamentales de la Programación Orientada a Objetos (POO) son cuatro conceptos clave: **Herencia**, **Polimorfismo**, **Encapsulamiento** y **Abstracción**. Estos principios permiten escribir código más modular, reutilizable y fácil de mantener. A continuación, se explican en detalle estos conceptos y cómo se aplican en Python.

### 3.13.1 1. Herencia

La herencia es un mecanismo que permite crear nuevas clases a partir de otras clases ya existentes, heredando sus atributos y métodos. Esto fomenta la reutilización del código, ya que permite evitar la duplicación de funciones comunes en varias clases.
Cuando una clase hereda de otra, se dice que la primera es la clase hija o subclase, y la segunda es la clase padre o superclase. La subclase puede extender o modificar el comportamiento de la clase padre, agregando nuevos métodos o sobrescribiendo métodos existentes.
**Ejemplo de herencia en Python:**

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

    def mostrar_informacion(self):
        print(f"Nombre: {self.nombre}, Edad: {self.edad}")

# La clase Estudiante hereda de Persona
class Estudiante(Persona):
    def __init__(self, nombre, edad, curso):
        super().__init__(nombre, edad)  # Llama al constructor de la clase padre
        self.curso = curso

    def mostrar_informacion(self):
        super().mostrar_informacion()
```

```
        print(f"Curso: {self.curso}")

# Uso de la herencia
estudiante1 = Estudiante("Juan", 20, "Matemáticas")
estudiante1.mostrar_informacion()
```

En este ejemplo, la clase `Estudiante` hereda de la clase `Persona`, y puede utilizar tanto el constructor como los métodos de la clase padre. Además, se puede agregar funcionalidad adicional (el atributo `curso` y su respectiva lógica).

### 3.13.2 2. Polimorfismo

El polimorfismo es la capacidad que tienen los objetos de diferentes clases de responder al mismo método de manera distinta. En POO, el polimorfismo permite tratar objetos de distintas clases de manera uniforme, siempre que estas clases compartan una interfaz común o implementen los mismos métodos.
El polimorfismo puede lograrse de dos formas principales:

- Polimorfismo en tiempo de ejecución (a través de la herencia).
- Polimorfismo de métodos (sobrecarga de métodos).

**Ejemplo de polimorfismo en Python:**

```
class Animal:
    def sonido(self):
        pass

class Perro(Animal):
    def sonido(self):
        print("El perro ladra.")

class Gato(Animal):
    def sonido(self):
        print("El gato maúlla.")

# Uso de polimorfismo
animales = [Perro(), Gato()]
for animal in animales:
    animal.sonido()
```

En este ejemplo, tanto `Perro` como `Gato` son subclases de `Animal` y ambos implementan el método `sonido`, pero cada uno lo hace de manera distinta. El mismo método se comporta de forma diferente según el tipo de objeto que lo invoque.

### 3.13.3 3. Encapsulamiento

El encapsulamiento consiste en restringir el acceso directo a los atributos y métodos de un objeto, permitiendo controlarlo mediante métodos definidos, llamados getters y setters. Esto

ayuda a proteger la integridad de los datos del objeto y garantiza que solo se pueda acceder a ellos de manera controlada.
En Python, el encapsulamiento se implementa utilizando convenciones de nombres:

- Un atributo precedido por un guion bajo (_) sugiere que es de uso interno, pero aún es accesible desde fuera de la clase (es solo una convención).
- Un atributo precedido por dos guiones bajos (__) activa la mangling de nombres, que hace más difícil su acceso desde fuera de la clase, aunque sigue siendo posible.

**Ejemplo de encapsulamiento en Python:**

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

    def mostrar_saldo(self):
        print(f"Titular: {self.titular}, Saldo: ${self.__saldo}")

    def depositar(self, cantidad):
        if cantidad > 0:
            self.__saldo += cantidad
        else:
            print("El depósito debe ser positivo.")

    def retirar(self, cantidad):
        if 0 < cantidad <= self.__saldo:
            self.__saldo -= cantidad
        else:
            print("Fondos insuficientes o cantidad inválida.")

# Uso del encapsulamiento
cuenta = CuentaBancaria("Ana", 1000)
cuenta.mostrar_saldo()
cuenta.depositar(500)
cuenta.retirar(200)
cuenta.mostrar_saldo()
```

En este ejemplo, el atributo `__saldo` está encapsulado, lo que significa que no puede ser modificado directamente desde fuera de la clase. Los métodos `depositar` y `retirar` permiten modificar el saldo de manera controlada.

### 3.13.4 4. Abstracción

La abstracción es el proceso de ocultar los detalles complejos de implementación y mostrar solo las funcionalidades esenciales de un objeto. En POO, las clases abstractas y los métodos abstractos permiten definir una estructura general que será implementada por otras clases concretas.

Python proporciona abstracción mediante la clase base `ABC` (Abstract Base Class) y el decorador `@abstractmethod`, que se utiliza para declarar métodos abstractos que deben ser implementados por las subclases.

**Ejemplo de abstracción en Python:**

```python
from abc import ABC, abstractmethod

class Forma(ABC):
    @abstractmethod
    def calcular_area(self):
        pass

class Rectangulo(Forma):
    def __init__(self, ancho, alto):
        self.ancho = ancho
        self.alto = alto

    def calcular_area(self):
        return self.ancho * self.alto

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

    def calcular_area(self):
        return 3.14 * self.radio * self.radio

# Uso de abstracción
formas = [Rectangulo(5, 3), Circulo(7)]
for forma in formas:
    print(f"Área: {forma.calcular_area()}")
```

En este ejemplo, la clase `Forma` es abstracta y define un método abstracto `calcular_area`. Las clases `Rectangulo` y `Circulo` implementan el método `calcular_area` de manera específica para sus formas geométricas. De esta manera, se ocultan los detalles de implementación de cada forma, y solo se expone la funcionalidad de calcular el área.

## 3.14 Resumen

- **Herencia:** Permite reutilizar y extender clases existentes.
- **Polimorfismo:** Permite que diferentes clases respondan al mismo método de manera distinta.
- **Encapsulamiento:** Restringe el acceso directo a los atributos y métodos, protegiendo la integridad del objeto.

- **Abstracción:** Oculta los detalles complejos de implementación, exponiendo solo las funcionalidades esenciales.

Estos conceptos son fundamentales para diseñar sistemas complejos y escalables utilizando el paradigma orientado a objetos en Python.

## Ejercicios de Herencia

### Ejercicio 1: Sistema de Gestión de Vehículos

Crea un programa que implemente una clase base **Vehiculo** con los atributos `marca` y `modelo`, y un método para mostrar la información del vehículo. Luego, crea dos clases hijas: **Auto** y **Moto**, que hereden de **Vehiculo**. La clase **Auto** debe tener un atributo adicional `puertas` y la clase **Moto** debe tener un atributo adicional `tipo_moto`. Ambas clases deben sobreescribir el método para mostrar la información, incluyendo sus atributos específicos.

### Ejercicio 2: Sistema de Gestión de Animales

Escribe un programa que implemente una clase base **Animal** con un método `sonido` que no haga nada (puede estar vacío o levantar una excepción). Luego, crea clases hijas **Perro**, **Gato**, y **Vaca**, que hereden de **Animal**, cada una con su propio método `sonido` que imprima el sonido característico del animal. Usa polimorfismo para crear una lista de animales y ejecutar el método `sonido` para cada uno.

### Ejercicio 3: Sistema de Control de Empleados

Crea una clase base **Empleado** que tenga los atributos `nombre` y `sueldo`, y un método para mostrar la información del empleado. Luego, crea dos clases hijas: **EmpleadoTiempoCompleto** y **EmpleadoMedioTiempo**. La clase **EmpleadoTiempoCompleto** debe tener un método para calcular el salario mensual (asumiendo un sueldo fijo), y la clase **EmpleadoMedioTiempo** debe tener un método para calcular el salario basado en el número de horas trabajadas.

### Ejercicio 4: Sistema de Gestión de Figuras Geométricas

Desarrolla una clase base **Figura** que tenga un método abstracto `calcular_area` que será implementado por las clases hijas. Crea clases **Rectangulo**, **Triangulo** y **Circulo** que hereden de **Figura**, cada una con su propia implementación del método `calcular_area`. El programa debe permitir crear varias figuras y calcular su área.

### Ejercicio 5: Sistema de Reserva de Transporte

Escribe un programa que implemente una clase base **Transporte** con los atributos `capacidad` y `tipo_combustible`. Crea clases hijas **Autobus**, **Taxi**, y **Bicicleta** que hereden de **Transporte**. Cada clase hija debe tener un método específico: por ejemplo, **Autobus** podría tener un método para calcular el costo por pasajero, **Taxi** un método para calcular el costo por kilómetro recorrido, y **Bicicleta** un método para mostrar la duración estimada de un viaje.

## 3.15 Ejercicios de Polimorfismo

**Ejercicio 1: Sistema de Pago de Empleados**

Crea una clase base **Empleado** con un método `calcular_salario()`. Luego, crea clases hijas **EmpleadoTiempoCompleto** y **EmpleadoMedioTiempo**, cada una con su propia implementación del método `calcular_salario()`. Usa polimorfismo para crear una lista de empleados y calcular el salario de cada uno, llamando al mismo método `calcular_salario()`.

**Ejercicio 2: Sistema de Gestión de Impuestos**

Escribe un programa que implemente una clase base **Producto** con un método `calcular_impuesto()`. Luego, crea clases hijas **ProductoElectronico**, **ProductoAlimenticio** y **ProductoRopa**, cada una con una implementación diferente del método `calcular_impuesto()`. Usa polimorfismo para aplicar el cálculo de impuestos a una lista de productos de diferentes tipos.

**Ejercicio 3: Sistema de Vehículos de Transporte**

Crea una clase base **Transporte** con un método `calcular_costo_viaje(distancia)`. Luego, crea clases hijas **Coche**, **Autobus** y **Bicicleta**, cada una con su propia implementación del método para calcular el costo del viaje en función de la distancia. Usa polimorfismo para calcular el costo de viaje de una lista de diferentes medios de transporte.

**Ejercicio 4: Sistema de Facturación de Servicios**

Desarrolla una clase base **Servicio** con un método `calcular_costo()`. Luego, crea clases hijas **ServicioInternet**, **ServicioTelefonico** y **ServicioTelevision**, cada una con su propia implementación del método `calcular_costo()`. Usa polimorfismo para calcular el costo mensual de diferentes servicios contratados por un cliente.

**Ejercicio 5: Sistema de Dibujar Formas**

Crea una clase base **Forma** con un método `dibujar()`. Luego, crea clases hijas **Rectangulo**, **Circulo** y **Triangulo**, cada una con su propia implementación del método `dibujar()`. Usa polimorfismo para crear una lista de formas y dibujar cada una llamando al mismo método `dibujar()`.

# 4 Algoritmía orientada a objetos con UML

## 4.1 Introducción al UML

### 4.1.1 Definición del UML

El Lenguaje de Modelado Unificado (UML) es un lenguaje gráfico estándar utilizado para visualizar, especificar, construir y documentar los componentes de un sistema de software. Creado para proporcionar una representación abstracta y entendible del diseño de sistemas, UML permite a los desarrolladores y arquitectos de software modelar la estructura y comportamiento de aplicaciones complejas antes de implementarlas en código.

El UML no está vinculado a un lenguaje de programación específico, lo que lo convierte en una herramienta versátil aplicable a diferentes entornos de desarrollo, incluida la programación orientada a objetos (POO). Proporciona una forma clara de representar los elementos y relaciones entre ellos, facilitando la comunicación entre desarrolladores y partes interesadas no técnicas.

### 4.1.2 Historia y evolución del UML

El UML nació a mediados de la década de 1990 como respuesta a la necesidad de un lenguaje común y estandarizado para el modelado de software. Antes de su creación, existían varios métodos y enfoques para el modelado, como el método de Booch, el método de Jacobson (OMT) y el método de Rumbaugh, entre otros. Sin embargo, estos enfoques eran limitados y no proporcionaban una solución integral para el diseño y análisis de sistemas complejos.

En 1994, tres de los más influyentes creadores de metodologías de modelado, Grady Booch, Ivar Jacobson y James Rumbaugh, se unieron bajo el auspicio de la empresa Rational Software para desarrollar un lenguaje estándar que unificara las mejores prácticas de los métodos existentes.

Este esfuerzo culminó en la primera versión de UML en 1997, que posteriormente fue adoptada y gestionada por la Object Management Group (OMG), una organización que regula estándares tecnológicos.

Con el paso del tiempo, UML ha evolucionado, y su última versión, incluye más diagramas y especificaciones que permiten un modelado más detallado y preciso de los sistemas.

### 4.1.3 Importancia del UML en la ingeniería de software

El uso de UML en la ingeniería de software ha ganado gran relevancia, ya que proporciona un lenguaje común y estándar que facilita la comunicación entre equipos de desarrollo, clientes y otros interesados. Esta capacidad para transmitir ideas y diseños de manera precisa reduce errores y malentendidos durante las etapas iniciales del desarrollo de software.

El UML es particularmente útil en la Programación Orientada a Objetos (POO), ya que permite modelar tanto la estructura como el comportamiento de sistemas basados en objetos. Los diagramas de clases, uno de los diagramas más comunes en UML, representan las clases, atributos, métodos y relaciones entre ellas, que son elementos fundamentales en la POO.

Algunas de las principales razones para utilizar UML en la ingeniería de software incluyen:

- **Visualización clara:** UML ofrece una representación visual de los sistemas, lo que facilita la comprensión del funcionamiento y la arquitectura del software.
- **Especificación precisa:** El UML permite especificar los componentes del sistema de manera detallada, lo que ayuda a evitar ambigüedades en los requerimientos.
- **Construcción guiada:** Los diagramas UML actúan como una hoja de ruta para los desarrolladores, proporcionando un esquema claro para la implementación del software.
- **Documentación estandarizada:** El UML facilita la creación de documentación clara y precisa, útil tanto durante el desarrollo como en el mantenimiento del sistema.

### 4.1.4 Objetivos del UML

Los principales objetivos de UML en el desarrollo de software son:

- **Proporcionar un estándar:** UML establece un lenguaje común que es entendible por todos los miembros de un equipo, independientemente de su nivel técnico.
- **Mejorar la comunicación:** Al ser un lenguaje visual, facilita el intercambio de ideas y la colaboración entre los miembros del equipo de desarrollo y las partes interesadas.
- **Facilitar el diseño y análisis:** UML permite diseñar y analizar sistemas antes de que se implemente el código, lo que ayuda a identificar problemas de diseño en etapas tempranas.
- **Documentar el sistema:** Proporciona una base sólida para la documentación del sistema, lo que es esencial para el mantenimiento y evolución del software.

### 4.1.5 Ámbitos de aplicación del UML

El UML se utiliza en una amplia variedad de aplicaciones dentro de la ingeniería de software. Algunos de los ámbitos más comunes son:

- **Desarrollo de software orientado a objetos:** Es ampliamente utilizado para modelar sistemas basados en POO, ayudando a estructurar y diseñar las clases y sus relaciones.
- **Sistemas empresariales:** UML es empleado para modelar grandes sistemas de información en empresas, permitiendo una mejor planificación y gestión del desarrollo.
- **Desarrollo de sistemas embebidos:** En proyectos donde el hardware y software están integrados, UML ayuda a modelar las interacciones entre estos componentes.
- **Arquitectura de software:** UML se usa para diseñar la arquitectura de sistemas complejos, como aplicaciones web, sistemas distribuidos y servicios en la nube.

### 4.1.6 Ventajas del UML

- **Versatilidad:** UML puede aplicarse a distintos tipos de sistemas, independientemente de la tecnología utilizada para desarrollarlos.
- **Estándar internacional:** Al estar gestionado por la OMG (Object Management Grupo), UML es reconocido globalmente como un estándar para el modelado de software.
- **Mejora la calidad del diseño:** La representación gráfica del sistema permite identificar problemas de diseño temprano, lo que reduce los errores en las etapas de desarrollo y mantenimiento.
- **Facilita la reutilización del código:** Al proporcionar un modelo claro y estandarizado del sistema, es más fácil identificar componentes que pueden ser reutilizados en otros proyectos.

### 4.1.7 Conclusión

El UML ha revolucionado la forma en que los ingenieros de software diseñan y documentan sistemas complejos. Al proporcionar un conjunto de herramientas gráficas para modelar tanto la estructura como el comportamiento del software, ha facilitado una mejor comunicación entre las partes interesadas y ha mejorado la calidad del diseño del software. En particular, en el contexto de la programación orientada a objetos, UML es una herramienta invaluable para traducir ideas abstractas en un código bien estructurado y eficiente.

## 4.2 Fundamentos del UML

### 4.2.1 Propósito del UML

El Lenguaje de Modelado Unificado (UML) fue creado con el propósito de proporcionar un lenguaje estandarizado para visualizar, especificar, construir y documentar sistemas de software, especialmente en el contexto de la Programación Orientada a Objetos (POO). El UML facilita la comunicación entre todos los involucrados en el ciclo de vida del desarrollo de software, desde analistas hasta desarrolladores, permitiendo un entendimiento claro del sistema a lo largo de todas sus fases.

El propósito principal del UML es ofrecer una forma visual de representar los sistemas de software de manera abstracta. Esto ayuda a los desarrolladores a planificar y diseñar sus aplicaciones de forma más eficiente, identificando relaciones, componentes y comportamientos antes de la implementación en código. Aunque UML está muy relacionado con la POO, también puede aplicarse a otros paradigmas de programación.

### 4.2.2 Beneficios del uso de UML en el desarrollo de software

El uso de UML trae múltiples beneficios a lo largo del desarrollo del software. Entre los más destacados están:

- **Mejora la comunicación:** Al ser un lenguaje visual, UML permite que los equipos de desarrollo y las partes interesadas comprendan el diseño del sistema de manera rápida y clara, independientemente de su nivel técnico.
- **Facilita la planificación y diseño:** Los diagramas UML proporcionan una visión clara de cómo los componentes del sistema se relacionan entre sí, lo que facilita la identificación de posibles problemas en las etapas tempranas del desarrollo.
- **Soporte para la toma de decisiones:** UML ayuda a los arquitectos de software y desarrolladores a tomar decisiones fundamentadas sobre el diseño y estructura del sistema antes de comenzar a programar.
- **Reutilización del diseño:** Con UML, es posible modelar y documentar componentes reutilizables que pueden ser implementados en varios proyectos, optimizando el tiempo y los recursos.
- **Documentación clara y estandarizada:** A medida que el proyecto avanza, los diagramas UML actúan como una documentación viva, lo que facilita el mantenimiento y la evolución del sistema.
- **Independencia de la tecnología:** UML no está vinculado a un lenguaje de programación específico, lo que lo hace útil en diferentes entornos de desarrollo, desde lenguajes orientados a objetos como Python o Java, hasta lenguajes procedurales.

### 4.2.3 Principios básicos de Modelado

El modelado con UML sigue algunos principios fundamentales que aseguran que los diagramas sean consistentes, comprensibles y útiles durante el proceso de desarrollo. Estos principios son:

- **Abstracción:** El modelado en UML se basa en la abstracción, lo que significa que se representan solo los aspectos clave del sistema. En lugar de incluir todos los detalles, los diagramas UML se enfocan en lo que es más relevante para la comprensión del diseño general.
- **Modularidad:** Los sistemas se dividen en componentes más pequeños y manejables. Esta división se refleja en UML mediante diferentes tipos de diagramas que permiten modelar aspectos específicos del sistema, como la estructura, el comportamiento o las interacciones entre los objetos.

- **Reusabilidad:** UML permite identificar patrones de diseño y componentes que pueden ser reutilizados en otros proyectos o dentro del mismo sistema. Este principio está alineado con las mejores prácticas de la POO.
- **Consistencia:** Los diagramas UML deben ser consistentes entre sí. Por ejemplo, las relaciones entre clases en un diagrama de clases deben reflejarse de manera coherente en otros diagramas, como los diagramas de secuencia o de casos de uso.
- **Evolutividad:** A medida que un sistema crece o cambia, el modelo UML puede ajustarse para reflejar esas modificaciones. UML proporciona una base flexible que permite la evolución del sistema sin perder claridad o integridad.

### 4.2.4 Los diagramas de UML y su rol en el desarrollo de software

El UML incluye una serie de diagramas que se agrupan en dos categorías principales: diagramas estructurales y diagramas de comportamiento. Cada tipo de diagrama juega un rol específico en la representación de diferentes aspectos del sistema:

- Diagramas estructurales: Estos diagramas muestran cómo están organizados los elementos del sistema y sus relaciones.
  - Diagrama de clases: Representa las clases y sus relaciones.
  - Diagrama de objetos: Modela instancias específicas de clases.
  - Diagrama de paquetes: Agrupa clases y componentes en módulos lógicos.
- Diagramas de comportamiento: Estos diagramas modelan cómo interactúan los componentes del sistema y cómo se comportan en respuesta a eventos.
  - Diagrama de casos de uso: Describe la interacción entre actores externos y el sistema.
  - Diagrama de secuencia: Muestra la interacción entre objetos a lo largo del tiempo.
  - Diagrama de actividades: Representa el flujo de trabajo o de procesos dentro del sistema.

Cada diagrama UML tiene un propósito específico y ofrece una perspectiva única del sistema, permitiendo que los equipos de desarrollo lo analicen desde diferentes ángulos. En conjunto, estos diagramas ofrecen una visión holística del diseño y funcionamiento del sistema, asegurando que se comprenda antes de ser implementado en código.

### 4.2.5 Aplicación de UML en la programación orientada a objetos

En la Programación Orientada a Objetos (POO), UML es particularmente útil, ya que permite modelar conceptos esenciales como las clases, objetos, herencia, polimorfismo, encapsulamiento, y abstracción. Mediante diagramas de clases, es posible identificar claramente las relaciones entre las clases y cómo interactúan los objetos en el sistema.

El proceso de diseño utilizando UML en el contexto de POO sigue el siguiente flujo:

1. Identificación de los requisitos del sistema: Utilizando diagramas de casos de uso, se identifican los actores y los casos de uso que describen lo que el sistema debe hacer.

2. Modelado de clases: A partir de los requisitos, se crean diagramas de clases que representan los objetos, atributos y métodos que compondrán el sistema.

3. Diseño de interacciones: Los diagramas de secuencia y comunicación se utilizan para modelar cómo los objetos interactúan entre sí durante la ejecución del sistema.

4. Especificación del comportamiento: Diagramas de actividades y estados se emplean para modelar los flujos de trabajo y la lógica interna del sistema.

Esta metodología basada en UML ayuda a los desarrolladores a planificar detalladamente los sistemas orientados a objetos y facilita la transición del diseño al código en lenguajes como Python.

### 4.2.6 Conclusión

Los fundamentos del UML proporcionan una base sólida para el modelado de sistemas de software complejos. Al entender los principios básicos de abstracción, modularidad, consistencia y reusabilidad, los desarrolladores pueden aplicar UML de manera efectiva en la programación orientada a objetos. Además, el uso de diagramas estructurales y de comportamiento permite que el diseño sea claro y comprensible antes de proceder con la implementación, lo que mejora significativamente la calidad y mantenibilidad del software.

## 4.3 Diagramas en UML

### 4.3.1 Tipos de diagramas UML: Estructurales y de comportamiento

El Lenguaje de Modelado Unificado (UML) es conocido por su capacidad para representar de manera visual los diferentes aspectos de un sistema de software. Para hacerlo, UML se divide en dos grandes categorías de diagramas: los diagramas estructurales y los diagramas de comportamiento. Cada uno de estos grupos se utiliza para modelar diferentes perspectivas del sistema, permitiendo un análisis más completo y detallado del diseño.

#### Diagramas estructurales

Los diagramas estructurales en UML se enfocan en representar la arquitectura estática de un sistema. Estos diagramas describen los elementos que componen el sistema (clases, objetos, componentes, etc.) y sus relaciones entre sí. No muestran cómo los elementos interactúan o cambian en el tiempo, sino que se centran en cómo están organizados y vinculados.

Entre los diagramas estructurales más importantes se encuentran:

- Diagrama de Clases: Este es uno de los diagramas más utilizados en UML, ya que muestra las clases de un sistema, sus atributos y métodos, así como las relaciones entre ellas. Es fundamental en la programación orientada a objetos, pues permite visualizar cómo las clases interactúan y colaboran para cumplir los objetivos del sistema.

- Diagrama de Objetos: Similar al diagrama de clases, pero en lugar de representar las clases como entidades abstractas, muestra instancias específicas de esas clases (objetos) y sus valores en un momento dado. Este diagrama es útil para ver ejemplos concretos de cómo las clases son instanciadas en el sistema.

- Diagrama de Componentes: Este diagrama modela la arquitectura física del sistema, mostrando cómo los diferentes módulos o componentes de software se organizan y se relacionan entre sí. Es particularmente útil para representar sistemas grandes que están compuestos por muchos componentes interdependientes.
- Diagrama de Paquetes: Ayuda a organizar el sistema dividiéndolo en módulos o paquetes lógicos. Los paquetes permiten agrupar clases y otros elementos del sistema de manera estructurada, facilitando la gestión de sistemas grandes y complejos.
- Diagrama de Estructura Compuesta: Este diagrama proporciona una vista detallada de la estructura interna de un componente o clase, mostrando cómo sus partes colaboran para realizar las funciones internas. Es particularmente útil para sistemas que incluyen múltiples subcomponentes dentro de clases.

**Diagramas de comportamiento**

A diferencia de los diagramas estructurales, los diagramas de comportamiento en UML se enfocan en modelar cómo los elementos del sistema interactúan entre sí y cómo se comportan a lo largo del tiempo. Estos diagramas muestran los flujos de trabajo, los estados y las interacciones entre los objetos o componentes del sistema en respuesta a eventos.

Algunos de los diagramas de comportamiento más utilizados son:

- Diagrama de Casos de Uso: Modela las interacciones entre actores externos (que pueden ser usuarios u otros sistemas) y el sistema. Muestra los casos de uso, que representan las funcionalidades que el sistema debe ofrecer a los actores. Es fundamental durante las fases iniciales de análisis, ya que ayuda a definir los requisitos del sistema desde la perspectiva del usuario.
- Diagrama de Secuencia: Este diagrama describe cómo los objetos del sistema interactúan entre sí en una secuencia temporal. Muestra los mensajes que se envían entre objetos para llevar a cabo una función específica, lo que lo hace útil para entender cómo los diferentes componentes del sistema colaboran en tiempo real.
- Diagrama de Actividades: Representa el flujo de trabajo o de proceso dentro del sistema, mostrando cómo se realizan las actividades y cómo fluyen entre ellas. Es especialmente útil para modelar procesos complejos que incluyen bifurcaciones y uniones, como las actividades de un sistema automatizado.
- Diagrama de Estados: Modela el ciclo de vida de un objeto en el sistema, mostrando los diferentes estados por los que pasa y los eventos que causan la transición entre esos estados. Es útil para modelar sistemas donde los objetos cambian de estado en respuesta a eventos, como en sistemas de control o aplicaciones basadas en eventos.
- Diagrama de Comunicación: Se centra en las interacciones entre los objetos y los mensajes que intercambian. Aunque es similar al diagrama de secuencia, se enfoca más en las relaciones entre los objetos que en la secuencia temporal de los mensajes.
- Diagrama de Temporización: Modela el comportamiento de los objetos o sistemas en función del tiempo, mostrando cómo cambian sus estados o interacciones a lo largo de un período de tiempo.

### 4.3.2 Introducción a los diagramas más importantes

Entre los diversos diagramas UML, algunos son considerados fundamentales para la mayoría de los sistemas y son frecuentemente utilizados en el desarrollo de software, especialmente en el contexto de la Programación Orientada a Objetos (POO). A continuación, se explican los diagramas más importantes y cómo se aplican en la práctica.

#### Diagrama de clases

El diagrama de clases es uno de los pilares del UML. Representa la estructura estática del sistema mostrando las clases, sus atributos, métodos y las relaciones entre ellas. En el contexto de la POO, este diagrama es esencial porque define el .esqueleto"del sistema, permitiendo a los desarrolladores planificar la interacción entre objetos y la jerarquía de herencia.

Un diagrama de clases típico incluye:

- Clases: Representadas por rectángulos divididos en tres secciones: nombre de la clase, atributos y métodos.
- Relaciones: Incluyen herencia (generalización/especialización), asociación (uso de clases), agregación (relación de todo a parte), y composición (relación fuerte de todo a parte).

#### Diagrama de casos de uso

El diagrama de casos de uso es crucial en la fase de análisis del sistema, ya que describe las interacciones que los usuarios o actores externos tienen con el sistema. Se utiliza para definir los requisitos funcionales desde la perspectiva del usuario. Este diagrama es clave para asegurar que el sistema proporcionará las funcionalidades correctas según las expectativas de los usuarios.

Un diagrama de casos de uso contiene:

- Actores: Representan a los usuarios o sistemas externos que interactúan con el sistema.
- Casos de uso: Representan las funcionalidades que el sistema debe proporcionar a los actores.
- Relaciones: Las líneas que conectan actores con los casos de uso, mostrando la interacción.

#### Diagrama de secuencia

El diagrama de secuencia es esencial para modelar el flujo de mensajes entre los objetos en el sistema en un orden temporal. Permite a los desarrolladores entender cómo los objetos colaboran para cumplir una función específica. Este diagrama es particularmente útil para identificar el orden de las llamadas a métodos y asegurar que el flujo de control es correcto.

En un diagrama de secuencia se muestran:

- Objetos: Representados en la parte superior del diagrama, cada objeto tiene una línea de vida que desciende verticalmente.
- Mensajes: Representados por flechas que conectan las líneas de vida de los objetos, indicando el intercambio de mensajes o llamadas a métodos.

#### Diagrama de actividades

El diagrama de actividades representa flujos de trabajo o procesos dentro de un sistema. Es útil cuando se necesita modelar procesos complejos con bifurcaciones y uniones. Este diagrama es excelente para modelar el flujo de actividades dentro de un sistema, como flujos de aprobación de un proceso empresarial.

En el diagrama de actividades se incluyen:

- Actividades: Representadas por rectángulos redondeados.
- Flujo de control: Flechas que muestran el orden en el que se ejecutan las actividades.
- Condiciones: Bifurcaciones que permiten la toma de decisiones durante el flujo de trabajo.

### 4.3.3 Conclusión

Los diagramas en UML son herramientas poderosas para representar y entender tanto la estructura como el comportamiento de un sistema de software. Mientras que los diagramas estructurales muestran cómo están organizados los componentes del sistema, los diagramas de comportamiento ayudan a visualizar las interacciones y el flujo de control. Conocer y dominar estos diagramas es esencial para cualquier desarrollador que desee implementar sistemas complejos de manera eficiente y efectiva.

## 4.4 Diagramas estructurales

Los diagramas estructurales en UML proporcionan una representación estática del sistema, describiendo cómo se organizan sus componentes y las relaciones entre ellos. Estos diagramas permiten ver la arquitectura del sistema desde diferentes perspectivas, facilitando el diseño y la planificación del software. A continuación, se explican algunos de los diagramas estructurales más importantes y su aplicación.

### 4.4.1 Diagrama de clases

#### Definición y elementos principales

El diagrama de clases es uno de los diagramas más importantes y utilizados en UML, especialmente en el contexto de la programación orientada a objetos (POO). Este diagrama describe la estructura estática del sistema, mostrando las clases, atributos, métodos y las relaciones entre ellas. Proporciona una visión clara de la organización de los objetos en el sistema y de cómo interactúan entre sí.

Un diagrama de clases contiene los siguientes elementos principales:

- Clases: Representadas por rectángulos divididos en tres secciones: el nombre de la clase (en la parte superior), los atributos de la clase (en la sección media) y los métodos o funciones de la clase (en la parte inferior).

- Atributos: Propiedades o características de la clase. En POO, los atributos definen el estado de un objeto.

- Métodos: Funciones o comportamientos que definen lo que puede hacer la clase. En POO, los métodos representan las acciones que puede realizar un objeto.

- Relaciones: Enlaces entre clases que muestran cómo se comunican o dependen unas de otras. Las relaciones pueden ser de varios tipos, como herencia, asociación, agregación y composición

**Asociación, agregación, composición y herencia**

- Asociación: Es una relación simple entre dos o más clases, que indica que los objetos de una clase pueden interactuar con los objetos de otra clase. En un diagrama de clases, se representa con una línea que conecta dos clases. Por ejemplo, un .ªlumno"puede estar asociado con una Çlase".

- Agregación: Es un tipo de asociación que indica una relación de "todo/parte", en la que una clase es un conjunto que contiene objetos de otras clases. Sin embargo, los objetos de las clases que forman parte del conjunto pueden existir independientemente. Se representa con una línea con un rombo vacío en el extremo de la clase que representa el "todo". Por ejemplo, una Üniversidad"tiene varias "Facultades", pero las facultades pueden existir independientemente de la universidad.

- Composición: Es una forma más fuerte de agregación en la que los objetos que forman parte de un conjunto no pueden existir independientemente de ese conjunto. Se representa con un rombo sólido. Por ejemplo, un Çoche.ºestá compuesto por un "Motor", y el motor no puede existir sin el coche.

- Herencia (Generalización/Especialización): Es una relación en la que una clase hereda los atributos y métodos de otra clase. Se representa con una línea con una flecha que apunta hacia la clase "padre". Por ejemplo, una clase "Vehículo"puede tener clases Çochez "Moto"que heredan sus características.

### 4.4.2 Diagrama de objetos

**Diferencia con el diagrama de clases**

El diagrama de objetos es similar al diagrama de clases, pero en lugar de representar clases como entidades abstractas, muestra instancias específicas de esas clases, es decir, los objetos. Mientras que el diagrama de clases define la estructura del sistema en términos generales (clases, atributos, métodos), el diagrama de objetos proporciona una instantánea de un momento particular en el sistema, mostrando cómo están configurados los objetos y sus relaciones en un punto específico del tiempo.

**Uso para representar instancias específicas**

El diagrama de objetos es útil cuando se desea modelar una configuración particular del sistema o cuando se necesita mostrar ejemplos concretos de cómo las clases se implementan como objetos en el sistema. Por ejemplo, si se tiene una clase "Persona", un diagrama de objetos puede mostrar instancias específicas como "Juan: Personaz "María: Persona", con sus atributos particulares (nombre, edad, etc.) asignados.
Los diagramas de objetos también son útiles para entender cómo los objetos interactúan entre sí en situaciones específicas, y para realizar pruebas de diseño, asegurando que las instancias funcionan como se espera.

### 4.4.3 Diagrama de paquetes

**Organización del sistema en módulos o paquetes**

El diagrama de paquetes permite organizar un sistema grande en módulos o paquetes lógicos, agrupando clases y otros elementos relacionados. Los paquetes ayudan a gestionar la complejidad dividiendo el sistema en partes más manejables y organizadas. Esta estructura modular facilita el mantenimiento, la evolución y la comprensión del sistema.
Un paquete en UML se representa como una carpeta y puede contener otros paquetes, clases u otros elementos. Las relaciones entre paquetes muestran cómo los diferentes módulos del sistema interactúan entre sí.
Por ejemplo, en un sistema de gestión universitaria, se podría tener un paquete llamado .Estudiantes", que contiene clases relacionadas con los estudiantes, como .Alumno", .Expedientez "Matriculación", y otro paquete llamado Çursos", que contiene clases como .Asignaturaz "Profesor".

### 4.4.4 Diagrama de componentes

**Representación de la arquitectura física del sistema**

El diagrama de componentes describe la arquitectura física del sistema de software, mostrando cómo se organizan los diferentes módulos de software (componentes) y cómo interactúan entre sí. Es especialmente útil en la fase de diseño de sistemas grandes y distribuidos, donde los diferentes componentes pueden estar separados físicamente (por ejemplo, en servidores diferentes).
Un componente representa una parte de software que tiene una función bien definida y puede ser desplegado de manera independiente. En el diagrama de componentes se muestran las relaciones entre estos componentes y cómo intercambian datos o servicios.
Por ejemplo, en una aplicación web, podría haber un componente llamado "Frontend"que representa la interfaz de usuario, y otro componente llamado "Backend"que maneja la lógica de negocio. Ambos componentes interactúan entre sí para cumplir con los requisitos del sistema.

### 4.4.5 Diagrama de estructura compuesta

**Modelado de la estructura interna de los componentes**

El diagrama de estructura compuesta es una herramienta útil para modelar la estructura interna de un componente o clase, mostrando cómo sus partes internas interactúan para

realizar sus funciones. Este diagrama permite ver los detalles de cómo está organizado un componente a nivel interno, lo que facilita el diseño y la comprensión de la arquitectura del sistema.
En este diagrama, se representan las partes internas del componente, los conectores entre estas partes, y los roles que cada parte desempeña dentro del componente. Es particularmente útil en sistemas complejos donde un solo componente puede estar compuesto por múltiples subcomponentes que colaboran entre sí.
Por ejemplo, un componente ”Coche” podría estar compuesto por partes internas como "Motor", "Transmisiónz "Sistema Eléctrico", y el diagrama de estructura compuesta mostraría cómo estas partes interactúan para hacer funcionar el coche.

## 4.5 Diagramas de comportamiento

Los diagramas de comportamiento en UML modelan cómo los elementos de un sistema interactúan y cambian con el tiempo. A diferencia de los diagramas estructurales, que se enfocan en la organización estática del sistema, los diagramas de comportamiento se centran en el flujo de acciones, eventos y la comunicación entre objetos a lo largo del tiempo. A continuación, se detallan algunos de los diagramas de comportamiento más importantes.

### 4.5.1 Diagrama de casos de uso

#### Representación de la interacción entre actores y el sistema

El diagrama de casos de uso es una herramienta fundamental para capturar los requisitos funcionales de un sistema desde la perspectiva del usuario o de un actor externo. Este diagrama muestra las interacciones entre los actores (usuarios u otros sistemas) y el sistema en sí, especificando qué funcionalidades (casos de uso) el sistema debe proporcionar a los actores.
El diagrama de casos de uso es especialmente útil durante la fase de análisis de un proyecto, ya que permite identificar claramente qué necesita hacer el sistema y cómo los usuarios se relacionan con él.

#### Elementos principales: Actores, casos de uso, relaciones

- Actores: Representan a los usuarios o sistemas externos que interactúan con el sistema. Pueden ser personas (usuarios finales) o entidades externas, como otros sistemas o bases de datos.

- Casos de Uso: Son las funcionalidades o servicios que el sistema ofrece a los actores. Un caso de uso describe una interacción específica entre un actor y el sistema, por ejemplo, Registrar Usuario.º "Procesar Pago".

- Relaciones: Las relaciones en el diagrama de casos de uso pueden ser de diferentes tipos:

  - Asociación: Conecta un actor con un caso de uso, indicando que el actor participa en esa funcionalidad.
  - Generalización: Relación jerárquica entre actores o entre casos de uso, donde un caso de uso o actor hereda el comportamiento de otro.

  - Inclusión: Indica que un caso de uso incluye el comportamiento de otro caso de uso como parte de su flujo.
  - Extensión: Permite que un caso de uso extienda la funcionalidad de otro en situaciones específicas.

  El diagrama de casos de uso es esencial para garantizar que los requisitos funcionales del sistema estén bien definidos y comprendidos tanto por los desarrolladores como por los usuarios.

### 4.5.2 Diagrama de secuencia

**Modelado de la interacción entre objetos en una secuencia temporal**

El diagrama de secuencia modela la interacción entre los objetos de un sistema a lo largo del tiempo. Representa cómo los objetos se comunican entre sí mediante el envío de mensajes en un orden temporal específico. Este diagrama es útil para visualizar el flujo de control y para entender cómo los métodos de diferentes objetos se invocan para lograr un objetivo específico. El diagrama de secuencia incluye los siguientes elementos:

- Objetos: Representados en la parte superior del diagrama con sus nombres subrayados. Cada objeto tiene una línea de vida que desciende verticalmente.
- Líneas de vida: Representan el tiempo durante el cual un objeto existe y está activo.
- Mensajes: Flechas horizontales que representan la comunicación entre objetos, como la llamada a un método o el envío de un mensaje. Los mensajes pueden ser síncronos (llamada directa a un método) o asíncronos (enviar una señal o mensaje).

Este diagrama es esencial para comprender el flujo de interacción entre objetos y asegurarse de que las secuencias de eventos estén alineadas con los requisitos del sistema.

### 4.5.3 Diagrama de actividades

**Representación de flujos de trabajo**

El diagrama de actividades representa los flujos de trabajo o procesos dentro de un sistema. Es una herramienta útil para modelar la lógica de los procesos de negocio o las operaciones dentro de un sistema de software. Este diagrama se centra en el flujo de control entre actividades y es útil cuando se requiere representar procesos con bifurcaciones, uniones o decisiones.
Los elementos principales del diagrama de actividades incluyen:

- Actividades: Representadas por rectángulos redondeados, son los pasos o tareas que se llevan a cabo en el proceso.
- Flujo de control: Flechas que conectan las actividades, indicando el orden en que se ejecutan.
- Decisiones: Representadas por diamantes, muestran bifurcaciones en el flujo del proceso, donde se toma una decisión para continuar por uno u otro camino.
- Inicio y fin: Los círculos que representan el inicio (círculo sólido) y el fin (círculo con un borde grueso) del flujo de trabajo.

Este diagrama es particularmente útil para visualizar flujos de trabajo complejos que incluyen condiciones, bucles y decisiones, ayudando a entender cómo las actividades se encadenan para completar un proceso.

### 4.5.4 Diagrama de estados

**Modelado del ciclo de vida de un objeto o sistema**

El diagrama de estados describe el ciclo de vida de un objeto o sistema, mostrando los diferentes estados por los que pasa un objeto y los eventos que causan la transición entre esos estados. Este diagrama es útil para modelar sistemas donde los objetos cambian su comportamiento en respuesta a eventos externos o internos, como sistemas de control o aplicaciones basadas en eventos.
Los elementos clave de este diagrama son:

- Estados: Representados por rectángulos con esquinas redondeadas, los estados describen la condición en la que se encuentra un objeto en un momento dado.
- Transiciones: Flechas que conectan los estados, indicando cómo un objeto cambia de un estado a otro. Las transiciones son desencadenadas por eventos o condiciones específicas.
- Eventos: Son los estímulos que causan que un objeto cambie de estado. Los eventos pueden ser externos (acciones del usuario) o internos (cambios en el sistema).

Este diagrama es útil para modelar objetos complejos que tienen múltiples estados posibles y pueden cambiar de estado en función de diversos eventos.

### 4.5.5 Diagrama de comunicación

**Interacción entre objetos, centrado en los mensajes que intercambian**

El diagrama de comunicación se enfoca en la interacción entre objetos en un sistema, destacando los mensajes que se intercambian entre ellos. Aunque tiene similitudes con el diagrama de secuencia, el diagrama de comunicación pone más énfasis en las relaciones entre los objetos y menos en el orden temporal de los mensajes.
En este diagrama, los objetos se representan como nodos, y los mensajes que intercambian se muestran como flechas entre estos nodos. Cada flecha indica el mensaje que un objeto envía a otro, y se numera para mostrar el orden en que se envían los mensajes.
El diagrama de comunicación es útil para visualizar las relaciones entre los objetos y cómo interactúan, lo que facilita el análisis de la estructura de comunicación en el sistema.

### 4.5.6 Diagrama de temporización

**Modelado del comportamiento de los objetos en el tiempo**

El diagrama de temporización se utiliza para modelar cómo cambian los estados de los objetos o sistemas en función del tiempo. Este diagrama muestra el comportamiento de los objetos en relación con una línea de tiempo, lo que es útil para sistemas que tienen restricciones temporales o donde los eventos dependen del paso del tiempo.
Los elementos clave incluyen:

- Líneas de tiempo: Representan el paso del tiempo, con marcas para eventos clave.
- Estados: Se muestran a lo largo de la línea de tiempo, indicando cómo el estado de un objeto cambia a lo largo del tiempo.
- Eventos: Son los puntos en el tiempo que desencadenan un cambio de estado o acción.

El diagrama de temporización es especialmente útil para modelar sistemas en tiempo real o sistemas en los que las respuestas deben ocurrir dentro de plazos específicos.

### 4.5.7 Conclusión

Los diagramas de comportamiento en UML son herramientas valiosas para modelar y entender cómo los elementos del sistema interactúan y cambian a lo largo del tiempo. Desde la representación de los requisitos funcionales mediante casos de uso hasta el modelado de interacciones y flujos de trabajo con diagramas de secuencia, actividades y estados, estos diagramas proporcionan una visión completa del comportamiento dinámico del sistema. Son esenciales para diseñar sistemas robustos y coherentes que respondan de manera adecuada a los eventos y flujos de trabajo definidos.

## 4.6 Elementos esenciales de los diagramas UML

Los diagramas UML (Lenguaje de Modelado Unificado) se basan en una serie de elementos fundamentales que permiten representar la estructura y el comportamiento de un sistema de software. Estos elementos se utilizan en una amplia variedad de diagramas y son esenciales para modelar los conceptos básicos de la Programación Orientada a Objetos (POO), así como otros aspectos de los sistemas. A continuación, se describen los principales elementos que componen los diagramas UML, junto con sus relaciones y uso.

### 4.6.1 Clases, atributos y métodos

Uno de los elementos más esenciales en UML es la clase, que representa un plano o plantilla a partir de la cual se crean los objetos. Las clases encapsulan datos (atributos) y comportamientos (métodos) relacionados, y son el núcleo de la programación orientada a objetos.

#### Clases

- Clase: Representada en UML por un rectángulo dividido en tres secciones. La parte superior muestra el nombre de la clase, la sección media lista los atributos y la parte inferior contiene los métodos. Una clase define las características y el comportamiento de los objetos que pertenecen a esa clase.
- Ejemplo: La clase Coche puede tener atributos como ”color” ”marca” y "modelo", y métodos como .ªcelerarz "frenar".

**Atributos**

- Atributos: Son las propiedades o características que describen el estado de una clase o de un objeto. En UML, los atributos se muestran en la segunda sección de la clase y se describen con su nombre y, opcionalmente, con su tipo y visibilidad (pública, privada, protegida).
- Ejemplo: Un atributo de la clase Coche podría ser color: String o marca: String.

**Métodos**

- Métodos: Son las funciones o acciones que una clase puede realizar. Los métodos definen el comportamiento de la clase, describiendo las operaciones que los objetos de la clase pueden ejecutar. En UML, los métodos se representan en la tercera sección de la clase y, al igual que los atributos, pueden incluir la visibilidad y el tipo de retorno.
- Ejemplo: Un método en la clase Coche podría ser acelerar() o frenar().

### 4.6.2 Relaciones entre clases

En UML, las relaciones entre las clases son fundamentales para describir cómo interactúan y se comunican los distintos componentes de un sistema. Las relaciones más comunes incluyen la herencia, dependencia, asociación, agregación y composición.

**Herencia**

- Herencia (Generalización/Especialización): Es una relación en la que una clase hija hereda los atributos y métodos de una clase padre. En UML, se representa con una línea sólida con una flecha hueca que apunta hacia la clase padre. La herencia es un concepto clave en la POO, ya que permite crear nuevas clases a partir de otras, reutilizando y extendiendo su comportamiento.
- Ejemplo: Una clase Vehículo puede tener dos clases derivadas llamadas Coche y Moto, que heredan los atributos y métodos de Vehículo.

**Dependencia**

- Dependencia: Es una relación débil que indica que una clase depende de otra para funcionar, pero no tiene una relación directa y constante con ella. Se representa con una línea discontinua con una flecha que apunta hacia la clase de la que depende. Esto significa que un cambio en la clase dependiente puede afectar la clase que depende de ella.
- Ejemplo: Una clase Pedido puede depender de una clase Producto, ya que necesita crear una instancia de Producto para procesar el pedido.

**Asociación**

- Asociación: Es una relación entre dos o más clases que indica que los objetos de una clase interactúan con los objetos de otra clase. Se representa con una línea simple entre las clases. Puede ser unidireccional o bidireccional.

- Ejemplo: Una clase Alumno puede estar asociada con una clase Curso, lo que indica que un alumno se inscribe en uno o más cursos

**Agregación y composición**

- Agregación: Es una relación especial de asociación que indica que una clase es un conjunto de otras clases, pero los objetos de las clases componentes pueden existir independientemente de la clase çontenedora". En UML, se representa con una línea y un rombo vacío en el extremo de la clase çontenedora".

- Ejemplo: Una clase Universidad puede tener varias Facultades, pero las facultades pueden existir sin la universidad.

- Composición: Es una forma más fuerte de agregación, donde los objetos de las clases componentes no pueden existir sin la clase contenedora. Se representa con un rombo sólido. La composición implica que el ciclo de vida de los componentes está ligado al de la clase contenedora.

- Ejemplo: Un Coche está compuesto por un Motor, y el motor no puede existir independientemente del coche.

### 4.6.3 Actores, objetos y componentes

Los actores, objetos y componentes son otros elementos esenciales en los diagramas UML que ayudan a describir el comportamiento y la estructura del sistema desde diferentes perspectivas.

**Actores**

- Actores: Representan a las personas, sistemas o entidades externas que interactúan con el sistema. Son elementos clave en los diagramas de casos de uso, ya que definen quién o qué está utilizando el sistema y qué acciones realiza.

- Ejemplo: En un sistema de reservas de vuelos, el actor puede ser el "Pasajero", quien interactúa con el sistema para buscar y reservar un vuelo.

**Objetos**

- Objetos: Son instancias concretas de las clases. Mientras que las clases son definiciones abstractas, los objetos son representaciones específicas que tienen valores particulares para sus atributos. En UML, los objetos se utilizan en diagramas como el diagrama de objetos y el diagrama de secuencia para mostrar interacciones en un momento determinado.

- Ejemplo: En una clase Coche, un objeto podría ser miCoche: Coche con atributos color = ”rojo” y marca = "Toyota".

**Componentes**

- Componentes: Son piezas modulares del sistema que representan un conjunto de funcionalidades agrupadas. Un componente en UML es una unidad de implementación que puede ser desplegada y reutilizada de forma independiente. Los componentes suelen aparecer en los diagramas de componentes para representar cómo los distintos módulos de software están organizados y conectados.
- Ejemplo: En una aplicación web, un componente podría ser el "Módulo de Autenticación", que maneja todo el proceso de inicio de sesión y registro de usuarios.

### 4.6.4 Conclusión

Los elementos esenciales de UML, como las clases, atributos, métodos, las diversas relaciones entre clases, y los actores, objetos y componentes, forman la base de los diagramas que modelan un sistema de software. Comprender estos elementos es fundamental para construir representaciones claras y precisas del sistema, permitiendo una mejor comunicación entre los desarrolladores y una transición más fluida entre el diseño y la implementación del software.

## 4.7 Herramientas para crear diagramas UML

Existen diversas herramientas de software que facilitan la creación y el diseño de diagramas UML. Estas herramientas ayudan a los desarrolladores y arquitectos de software a visualizar y modelar sistemas de manera clara y precisa. Algunas de ellas son gratuitas, mientras que otras son de pago y ofrecen funcionalidades más avanzadas, como la integración con sistemas de desarrollo y bases de datos. A continuación, se presentan algunas de las herramientas más populares para la creación de diagramas UML y una comparación de sus características.

### 4.7.1 Software disponible para modelado UML

A lo largo de los años, muchas herramientas se han desarrollado específicamente para el modelado UML. A continuación, se destacan algunas de las más conocidas y utilizadas en la industria:

**Enterprise Architect**

- Descripción: Enterprise Architect, desarrollado por Sparx Systems, es una herramienta de modelado UML ampliamente utilizada por empresas y desarrolladores. Es una de las soluciones más completas, ya que admite no solo UML, sino también otros lenguajes de modelado, como BPMN (Business Process Model and Notation) y SysML (Systems Modeling Language).
- Características:
  - Soporte completo para todos los tipos de diagramas UML.
  - Generación de código desde modelos UML.
  - Simulación de sistemas y análisis de comportamiento.
  - Colaboración en equipo y control de versiones.

  - Integración con plataformas de desarrollo como Visual Studio y Eclipse.

- Ventajas:
  - Completo soporte para empresas grandes con múltiples usuarios.
  - Amplia gama de características, desde diseño hasta generación de código y pruebas.

- Desventajas:
  - Es una herramienta de pago, y las licencias pueden ser costosas.
  - Requiere una curva de aprendizaje considerable debido a su amplio conjunto de características.

**Lucidchart**

- Descripción: Lucidchart es una herramienta en línea que permite crear diagramas UML de forma intuitiva. Es muy popular debido a su facilidad de uso y accesibilidad desde cualquier navegador, ya que no requiere instalación. Está orientada a usuarios que buscan una solución rápida y fácil de usar para la creación de diagramas.

- Características:
  - Herramienta basada en la web, accesible desde cualquier dispositivo con navegador.
  - Biblioteca de formas y elementos UML predefinidos.
  - Colaboración en tiempo real, con comentarios y edición por múltiples usuarios.
  - Integración con herramientas populares como Google Drive, Confluence y Jira.

- Ventajas:
  - Fácil de usar para diagramas rápidos y sencillos.
  - Ideal para la colaboración en equipos distribuidos.
  - Modelo de precios basado en suscripción, con una versión gratuita que incluye funcionalidades básicas.

- Desventajas:
  - No tiene capacidades avanzadas de generación de código o simulación de sistemas.
  - La versión gratuita tiene limitaciones en cuanto a funciones y cantidad de diagramas.

**Visual Paradigm**

- Descripción: Visual Paradigm es otra herramienta muy completa para la creación de diagramas UML. Además de UML, admite una amplia variedad de otros lenguajes de modelado y proporciona funciones avanzadas para generar y sincronizar código, pruebas de software y documentación.

- Características:

  - Soporte para UML, BPMN, ERD (Diagramas de Entidad-Relación) y otras notaciones.
  - Generación y sincronización de código en varios lenguajes de programación.
  - Soporte para trabajo en equipo y control de versiones.
  - Integración con plataformas de desarrollo como IntelliJ IDEA y Eclipse.
- Ventajas:
  - Soporte para una gran variedad de notaciones y técnicas de modelado.
  - Capacidades avanzadas de documentación y generación de código.
- Desventajas:
  - Es una herramienta de pago con modelos de suscripción y licencia.
  - Aunque tiene una interfaz fácil de usar, las características avanzadas pueden ser complicadas para los principiantes.

**StarUML**

- Descripción: StarUML es una herramienta de modelado de código abierto que soporta UML 2.x y ofrece un entorno ligero y accesible para la creación de diagramas. Es una solución popular para desarrolladores que prefieren software de código abierto.
- Características:
  - Soporte para diagramas UML y generación de código.
  - Interfaz sencilla y personalizable.
  - Soporte para plugins y extensiones.
- Ventajas:
  - Ligero y fácil de usar.
  - Compatible con sistemas operativos Windows, macOS y Linux.
  - Ideal para proyectos de pequeña y mediana escala.
- Desventajas:
  - No es tan completo como otras herramientas como Enterprise Architect o Visual Paradigm.
  - Las funciones avanzadas requieren instalar plugins.

**Draw.io**

- Descripción: Draw.io (ahora conocido como diagrams.net) es una herramienta de creación de diagramas en línea gratuita y fácil de usar. Aunque no está diseñada específicamente para UML, permite crear diagramas UML utilizando plantillas y bibliotecas de símbolos.
- Características:

  - Herramienta basada en la web que no requiere instalación.
  - Soporte para la creación de diagramas UML mediante bibliotecas de formas predefinidas.
  - Integración con plataformas como Google Drive, Dropbox y GitHub.
  - Gratuita y de código abierto.

- Ventajas:
  - Fácil de usar y accesible desde cualquier dispositivo.
  - Completamente gratuita y de código abierto.

- Desventajas:
  - No tiene soporte específico para UML avanzado, como generación de código o simulación.
  - No está tan orientada a grandes equipos o proyectos complejos.

### 4.7.2 Comparación de herramientas populares

| **Herramienta** | **Soporte UML Completo** | **Generación de Código** | **Colaboración** | **Facilidad de Uso** | **Licencia** |
|---|---|---|---|---|---|
| Enterprise Architect | Sí | Sí | Sí | Moderada | Pago |
| Lucidchart | Parcial | No | Sí | Alta | Freemium |
| Visual Paradigm | Sí | Sí | Sí | Moderada | Pago |
| StarUML | Sí | Sí | No | Alta | Pago |
| Draw.io | Parcial | No | Sí | Alta | Gratuita |

**Tabla 4.1:** Comparación de herramientas de modelado UML

**Recomendaciones:**

- Para grandes empresas o proyectos complejos: Enterprise Architect o Visual Paradigm son las opciones más adecuadas debido a su soporte completo para UML, generación de código y colaboración en equipo.

- Para equipos pequeños o usuarios individuales: StarUML es una opción sólida si se busca una herramienta ligera, mientras que Lucidchart es ideal para usuarios que necesitan diagramas rápidos y colaborativos sin funcionalidades avanzadas.

- Para uso gratuito y rápido: Draw.io es una excelente opción, especialmente para aquellos que buscan una solución accesible y fácil de usar para crear diagramas UML básicos.

### 4.7.3 Conclusión

La elección de una herramienta para crear diagramas UML dependerá de las necesidades específicas del proyecto, el tamaño del equipo y los requisitos técnicos. Herramientas como Enterprise Architect y Visual Paradigm ofrecen soluciones avanzadas para proyectos grandes y complejos, mientras que herramientas como Lucidchart, StarUML y Draw.io son más accesibles para equipos pequeños o proyectos de menor escala. Es importante seleccionar la herramienta adecuada que equilibre las funcionalidades que se necesitan con el presupuesto disponible.

## 4.8 UML y programación orientada a objetos

El Lenguaje de Modelado Unificado (UML) y la Programación Orientada a Objetos (POO) están intrínsecamente relacionados, ya que UML se diseñó para representar de manera gráfica los conceptos fundamentales de la POO, como clases, objetos, herencia y polimorfismo. En este contexto, UML actúa como una herramienta visual que facilita el diseño y la planificación de sistemas orientados a objetos, ayudando a los desarrolladores a conceptualizar y organizar su código antes de implementarlo.

### 4.8.1 Relación entre UML y POO

La Programación Orientada a Objetos (POO) es un paradigma de programación que organiza el código en objetos, los cuales son instancias de clases. Estos objetos interactúan entre sí para realizar las tareas definidas por el programa. Los conceptos clave de la POO incluyen clases, objetos, herencia, polimorfismo, encapsulamiento, y abstracción. UML, por su parte, es una herramienta visual que permite modelar estos conceptos de manera gráfica.
La relación entre UML y POO se basa en que UML proporciona una representación visual de las clases, objetos y relaciones entre ellos, permitiendo a los desarrolladores planificar la estructura de su código antes de implementarlo. Al utilizar UML, los desarrolladores pueden visualizar cómo se relacionan las clases, qué métodos y atributos contiene cada una, y cómo los objetos interactúan para cumplir con los requisitos del sistema.

Por ejemplo:

- Los diagramas de clases en UML representan la estructura estática de las clases, mostrando los atributos, métodos y relaciones, que luego se traducen directamente a las definiciones de clases en un lenguaje orientado a objetos como Python.

- Los diagramas de secuencia muestran la interacción entre los objetos en tiempo de ejecución, ayudando a planificar cómo se enviarán los mensajes y qué métodos se invocarán entre los objetos.

- Los diagramas de casos de uso permiten capturar los requisitos funcionales desde la perspectiva de los usuarios o actores, lo que ayuda a definir qué funcionalidades debe implementar el sistema.

### 4.8.2 Cómo los diagramas UML se traducen a código en Python

Uno de los beneficios más importantes de UML es su capacidad para traducir fácilmente los diagramas en código de lenguajes orientados a objetos como Python. A continuación, se describe cómo los principales diagramas UML se pueden transformar en código Python.

#### Diagrama de clases a código Python

El diagrama de clases es uno de los más utilizados en UML y puede ser traducido directamente a código en Python. Cada clase en el diagrama UML se convierte en una clase en Python, y los atributos y métodos que aparecen en el diagrama se implementan como variables de instancia y métodos en la clase Python.

Por ejemplo, dado el siguiente diagrama de clases en UML:

| **Coche** |
|---|
| marca : str<br>modelo : str<br>color : str |
| arrancar()<br>frenar() |

Este diagrama se puede traducir al siguiente código en Python:

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

    def arrancar(self):
        print(f"El {self.marca} {self.modelo} está arrancando.")

    def frenar(self):
        print(f"El {self.marca} {self.modelo} está frenando.")
```

En este ejemplo, el diagrama de clases se traduce directamente a código:

- Los atributos marca, modelo y color se convierten en las variables de instancia.
- Los métodos arrancar() y frenar() se implementan como funciones dentro de la clase.

#### Diagrama de secuencia a código Python

El diagrama de secuencia muestra cómo los objetos interactúan entre sí mediante el intercambio de mensajes en un orden temporal específico. Estos mensajes se traducen a llamadas a métodos entre objetos en Python. Por ejemplo, si un diagrama de secuencia

muestra la interacción entre dos objetos, Coche y Motor, para arrancar el coche, se puede traducir de la siguiente manera: **Diagrama de secuencia:**

| >Emisor | >Mensaje (acción) |
|---|---|
| Coche | `encender_motor()` → Motor |
| Motor | `confirmación()` → Coche |

**Código en Python:**

```python
class Motor:
    def encender_motor(self):
        print("Motor encendido.")
        return True

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

    def arrancar(self):
        if self.motor.encender_motor():
            print(f"El {self.marca} {self.modelo} ha arrancado.")
```

En este caso, el intercambio de mensajes entre Coche y Motor se traduce en una llamada al método encender_motor() de la clase Motor, y luego el coche recibe la confirmación y completa la acción de arrancar.

### Diagrama de casos de uso a código Python

El diagrama de casos de uso describe las interacciones entre los actores y el sistema, lo que ayuda a definir los requisitos funcionales. En Python, cada caso de uso puede traducirse a funciones o métodos que proporcionan la funcionalidad requerida.
Por ejemplo, si en un diagrama de casos de uso un actor llamado Üsuarioïnteractúa con un sistema para Ïniciar Sesión", esto se traduce a un método como el siguiente:

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

    def iniciar_sesion(self, contrasena):
        if self.contrasena == contrasena:
            print(f"Bienvenido, {self.nombre}.")
        else:
            print("Contraseña incorrecta.")
```

Aquí, el caso de uso Ïniciar Sesión"define que el usuario debe proporcionar una contraseña para acceder al sistema, y esta funcionalidad se implementa en el método iniciar_sesion().

## 4.9 Ejemplos:

A continuación presentamos ejemplos prácticos para aplicar los conceptos aprendidos sobre UML y su relación con la Programación Orientada a Objetos (POO). Estos ejercicios te permitirán crear diagramas UML para ejemplos sencillos y modelar un sistema orientado a objetos utilizando UML.

### 4.9.1 Ejemplo 1: Sistema de gestión de biblioteca

Supongamos que tenemos un proyecto que consiste en diseñar un sistema de gestión de biblioteca. A continuación se muestra cómo los diagramas UML se pueden implementar en el desarrollo de este sistema.

1. **Diagrama de clases:** El diagrama de clases para el sistema podría incluir clases como Libro, Usuario, y Biblioteca. En UML, estas clases y sus relaciones se representarán gráficamente, y luego se traducirán a código Python.
   **Diagrama de clases: (simplificado)**

| **Libro** |
|---|
| título : str |
| autor : str |
| isbn : str |
| disponible : bool |
| prestar_libro() |
| devolver_libro() |

| **Usuario** |
|---|
| nombre : str |
| id_usuario : str |
| tomar_prestado(libro : Libro) |
| devolver(libro : Libro) |

   **Código Python:**

```python
class Libro:
    def __init__(self, titulo, autor, isbn):
        self.titulo = titulo
        self.autor = autor
```

```
        self.isbn = isbn
        self.disponible = True

    def prestar_libro(self):
        if self.disponible:
            self.disponible = False
            print(f"El libro '{self.titulo}' ha sido prestado.")
        else:
            print(f"El libro '{self.titulo}' no está disponible.")

    def devolver_libro(self):
        self.disponible = True
        print(f"El libro '{self.titulo}' ha sido devuelto.")

class Usuario:
    def __init__(self, nombre, id_usuario):
        self.nombre = nombre
        self.id_usuario = id_usuario

    def tomar_prestado(self, libro):
        libro.prestar_libro()

    def devolver(self, libro):
        libro.devolver_libro()
```

2. **Diagrama de Secuencia:** El diagrama de secuencia para el proceso de prestar un libro mostrará la interacción entre los objetos Usuario y Libro para ejecutar la operación.

   **Secuencia:**

   El objeto Usuario llama al método tomar_prestado(libro) en el objeto Libro.
   El objeto Libro verifica si está disponible y cambia su estado.

   Este es un ejemplo básico de cómo los diagramas UML pueden guiar el desarrollo del código en Python para un sistema de gestión de biblioteca.

### 4.9.2 Ejemplo 2: Diagrama de clases para una biblioteca

Se te pide crear un diagrama de clases para un sistema de gestión de una biblioteca. El sistema debe ser capaz de gestionar libros y usuarios. Los libros pueden ser prestados y devueltos por los usuarios.

**Requisitos:**

1. Cada libro tiene un título, autor, ISBN y estado (disponible/no disponible).
2. Un usuario puede prestar y devolver libros.

3. Cada usuario tiene un nombre y un identificador único.

**Instrucciones:**

1. Identifica las clases necesarias para el sistema.
2. Define los atributos y métodos para cada clase.
3. Establece las relaciones entre las clases (asociación, agregación o composición).

**Solución:** El sistema tendrá al menos dos clases: Libro y Usuario.
Los atributos y métodos pueden estar definidos de la siguiente manera:
**Diagrama de clases: (simplificado)**

| **Libro** | **Usuario** |
|---|---|
| título : str<br>autor : str<br>isbn : str<br>disponible : bool | nombre : str<br>id_usuario : str |
| prestar_libro()<br>devolver_libro() | tomar_prestado(libro : Libro)<br>devolver(libro : Libro) |

En este diagrama:

- La asociación entre Usuario y Libro indica que un usuario puede prestar uno o más libros.
- Los métodos prestar_libro() y devolver_libro() en la clase Libro controlan el estado del libro (si está disponible o no).
- Los métodos tomar_prestado() y devolver() en la clase Usuario permiten a los usuarios interactuar con los libros.

### 4.9.3 Ejemplo 3: Diagrama de secuencia para el proceso de compra en una tienda en línea

Modela un diagrama de secuencia que describa el proceso de compra en una tienda en línea. El proceso incluye:

1. El usuario selecciona un producto.
2. El sistema de pagos procesa el pago.
3. Una confirmación se envía al usuario.

**Instrucciones:**

1. Identifica los objetos involucrados (por ejemplo: Usuario, Producto, SistemaPagos).

2. Representa los mensajes que se envían entre los objetos en el orden correcto.
3. Muestra el flujo de interacción.

**Solución:**

**Objetos involucrados:**

- Usuario
- Producto
- SistemaPagos

**Flujo de interacción:**

1. El Usuario selecciona un Producto.
2. El SistemaPagos procesa el pago del usuario.
3. El SistemaPagos envía una confirmación de pago al Usuario.

**Diagrama de Secuencia (Simplificado):**

| | | |
|---|---|---|
| Usuario | → | Producto: seleccionar_producto() |
| Usuario | → | SistemaPagos : procesar_pago() |
| SistemaPagos | → | Usuario : enviar_confirmacion() |

En este diagrama de secuencia:

- El Usuario interactúa primero con el Producto para seleccionarlo.
- Luego, el Usuario envía una solicitud al SistemaPagos para procesar el pago.
- Finalmente, el SistemaPagos envía una confirmación al Usuario indicando que la transacción fue exitosa.

### 4.9.4 Ejemplo 4: Sistema de gestión de estudiantes

Crea un modelo UML para un sistema de gestión de estudiantes en una universidad. El sistema debe permitir a los profesores gestionar las calificaciones de los estudiantes y a los estudiantes consultar sus calificaciones.

**Requisitos:**

1. Cada estudiante tiene un nombre, un identificador de estudiante y un conjunto de calificaciones.
2. Cada profesor puede asignar calificaciones a los estudiantes.
3. Los estudiantes pueden ver sus calificaciones.
4. Las asignaturas están vinculadas a los estudiantes y a los profesores.

**Instrucciones:**

1. Crea un diagrama de clases para representar los estudiantes, profesores y asignaturas.
2. Define las relaciones entre las clases (asociaciones, agregaciones, etc.).
3. Crea un diagrama de casos de uso para describir cómo los actores interactúan con el sistema.

**Solución:**

- Clases: Estudiante, Profesor, Asignatura.
- Atributos:
  - Estudiante: nombre, id_estudiante, calificaciones.
  - Profesor: nombre, id_profesor.
  - Asignatura: nombre_asignatura, calificaciones_estudiantes.
- Relaciones:
  - Un Profesor tiene una asociación con una o más Asignaturas.
  - Un Estudiante está asociado a una o más Asignaturas.
  - La relación entre Estudiante y Asignatura incluye una composición, ya que las calificaciones están directamente relacionadas con una asignatura.

**Diagrama de Clases (Simplificado):**

| **Estudiante** |
|---|
| nombre: `str`<br>id_estudiante: `str`<br>calificaciones: `dict` |
| consultar_calificaciones() |

| **Profesor:** |
|---|
| nombre: `str`<br>id_profesor: `str` |
| asignar_calificacion (estudiante: Estudiante, calificacion: float) |

| **Asignatura** |
|---|
| nombre_asignatura: `str`<br>calificaciones_estudiantes: `dict` |

**Diagrama de Casos de Uso (Simplificado):**

| **Actores** |
|---|
| Estudiante |
| Profesor |

| Casos de Uso: |
| --- |
| 1. Estudiante consulta sus calificaciones. |
| 2. Profesor asigna calificaciones a estudiantes. |

| Relaciones: |
| --- |
| El Profesor puede asignar calificaciones a los estudiantes en las asignaturas que enseña. |
| El Estudiante puede consultar sus calificaciones en las asignaturas en las que está inscrito. |

**Explicación:**

- La clase Profesor puede asignar calificaciones a los estudiantes en una asignatura utilizando el método asignar_calificacion().
- La clase Estudiante puede consultar sus calificaciones en cada asignatura mediante el método consultar_calificaciones().
- La clase Asignatura contiene las calificaciones asociadas a los estudiantes, lo que representa la composición entre la relación de los estudiantes y las asignaturas.

## 4.10 Ejercicios propuestos

### 4.10.1 Ejercicio 1: Sistema de gestión de inventario

Crea un diagrama de clases para un sistema de gestión de inventario en una tienda. El sistema debe gestionar productos, cada uno con un nombre, categoría, precio y cantidad en stock.

**Requisitos:**

1. Cada producto tiene un nombre, categoría, precio y cantidad en stock.
2. Los empleados pueden actualizar la cantidad de productos en el inventario.
3. Los clientes pueden consultar la disponibilidad de productos.

**Instrucciones:**

1. Identifica las clases necesarias.
2. Crea un diagrama de clases que incluya los atributos y métodos necesarios.
3. Establece las relaciones entre las clases.

### 4.10.2 Ejercicio 2: Sistema de reserva de hoteles

Crea un diagrama de casos de uso para un sistema de reservas en un hotel. El sistema debe permitir a los clientes realizar reservas, y a los administradores gestionar las habitaciones y consultar las reservas.

**Requisitos:**

1. Los clientes pueden buscar habitaciones disponibles y realizar una reserva.
2. Los administradores pueden añadir o eliminar habitaciones y consultar las reservas hechas por los clientes.

**Instrucciones:**

1. Identifica los actores y los casos de uso necesarios para este sistema.
2. Crea un diagrama de casos de uso que muestre cómo interactúan los actores con el sistema.
3. Describe las relaciones entre los actores y los casos de uso.

### 4.10.3 Ejercicio 3: Sistema de pedido de comida en línea

Modela un diagrama de secuencia que represente el proceso de realizar un pedido de comida en línea. El sistema debe permitir que el cliente seleccione los platos, realice el pago y reciba una confirmación del restaurante.

**Requisitos:**

1. El cliente selecciona platos de un menú.
2. El cliente realiza el pago.
3. El sistema envía una confirmación de pedido al cliente y al restaurante.

**Instrucciones:**

1. Identifica los objetos involucrados en este proceso.
2. Crea un diagrama de secuencia que muestre las interacciones entre los objetos en el orden correcto.
3. Asegúrate de incluir los mensajes que se envían entre los objetos.

### 4.10.4 Ejercicio 4: Sistema de registro de estudiantes

Diseña un diagrama de clases para un sistema de registro de estudiantes en una universidad. Los estudiantes pueden registrarse en cursos, y los profesores pueden asignar calificaciones a los estudiantes.

**Requisitos:**

1. Cada estudiante tiene un nombre, ID de estudiante y una lista de cursos inscritos.
2. Cada curso tiene un nombre, código de curso y un profesor asignado.
3. Los profesores pueden asignar calificaciones a los estudiantes en los cursos que imparten.

**Instrucciones:**

1. Identifica las clases necesarias para este sistema.
2. Crea un diagrama de clases que incluya los atributos y métodos para cada clase.
3. Establece las relaciones entre las clases.

### 4.10.5 Ejercicio 5: Sistema de gestión de proyectos

Crea un diagrama de clases para un sistema de gestión de proyectos. El sistema debe permitir que los empleados trabajen en proyectos, y cada proyecto puede tener varias tareas asignadas a diferentes empleados.

**Requisitos:**

1. Cada empleado tiene un nombre y un identificador.
2. Cada proyecto tiene un nombre, una descripción y una lista de tareas.
3. Cada tarea tiene un nombre, una descripción y un empleado asignado.
4. Los empleados pueden actualizar el estado de las tareas en las que están trabajando.

**Instrucciones:**

1. Crea un diagrama de clases que incluya las clases Empleado, Proyecto y Tarea.
2. Define las relaciones entre los empleados, los proyectos y las tareas.
3. Incluye los atributos y métodos necesarios para cada clase.

# 5

# Veinte ejemplos con algoritmía y lenguaje python orientados a objetos

Ejemplo No 1

Enunciado

| Calcular la suma de dos números reales, digitados por el usuario. |
|---|

1. IDENTIFICACION Y DELIMITACION DEL PROBLEMA A RESOLVER
   1.1 Identificación
   El enunciado es corto y trivial. Su interpretación no admite ambigüedades.

   1.2 Delimitación
   El problema está bien delimitado. Sólo es calcular la suma.

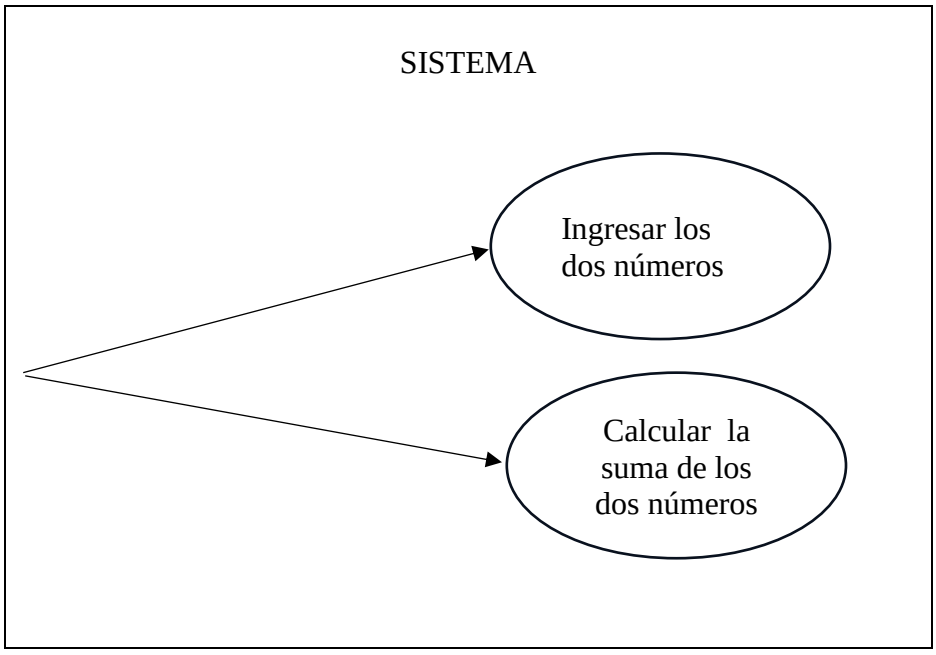
2. ANALISIS DEL PROBLEMA

   (Representación de casos de uso e identificación de clases, usando UML)

CASOS DE USO

---

UML: Lenguaje Unificado de Modelado Lenguaje gráfico para visualizar, especificar, construir y documentar los artefactos de un sistema de software.



2.1 Identificación de las clases

| Adición |
| --- |
| num1<br>num2 |
| sumar() |

s : Adición

2.1.1 Documentación de los miembros de la clase

num1 : Primer número. Tipo real.
num2 : Segundo número. Tipo real
sumar() : Método para sumar los dos números
s : Instancia u objeto de la clase Adicion

2.2 Herencia de clases y/o Polimorfismo

No se presenta herencia de clases.
No hay polimorfismo.

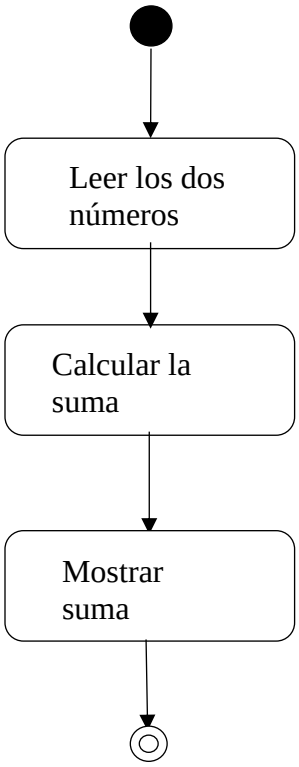
3. DESARROLLO DE LA SOLUCION

3.1 Modelación Matemática

Para este caso, se requiere un cálculo simple.
resultado = num1 + num2

3.2 Propuesta de solución

3.2.1 Diagrama de actividades




3.2.2 Definición de estructuras de datos

No existen.

3.2.3 Desarrollo del algoritmo (con diagramas NASSI – SCHEIDERMAN (N - S)):

El cual utiliza una simbología clara y visual, basada en rectángulos, por esto también se conoce como diagramación rectangular.

| Inicio | Operaciones |
|---|---|
| Declaración de clases, variables y constantes<br><br>Adición: s<br>num1<br>num2 | Adición :: sumar<br>retorne num1 + num2 |
| Algoritmo principal | |
| leer num1<br>leer num2<br>s = Adición(num1, num2)<br>resultado = s.sumar()<br>imprimir resultado | |

## 4. IMPLEMENTACIÓN

```
# -*- coding: utf-8 -*-
"""
Creado Octubre 25 de 2023
Nombre: manualPOOPython01
Que hace: Calcula la suma de dos números
Autores: Ms. Hernando Alvarez R.
         Ms. Leonardo Alvarez V.
         Ms. Luis Fernando Alvarez V.
Universidad Tecnológica de Pereira
"""
class Adicion: # Clase principal
    def __init__(self, num1, num2): # Método constructor
        self.num1 = num1            # en python
        self.num2 = num2
    def sumar(self): # Método sumar
        return self.num1 + self.num2

# Ejemplo de uso
def main():
    # Solicitar los números al usuario
    num1 = float(input("1er número_ "))
    num2 = float(input("2do número_ "))

    # Crear una instancia de la clase Adicion
    s = Adicion(num1,num2)

    # Obtener el resultado de la suma
    resultado = s.sumar()

    # Imprimir el resultado de la suma
    print("La suma de ",num1," y ",num2," es: ",resultado)
if "__name__ == __main__":
    main()

"""
self es una convención. No es una palabra reservada.
self es un parámetro en cada método
self representa la instancia de la clase
La sangría es la forma que usa python para
agrupar declaraciones
__name__ Es una variable integrada, que
evalúa el nombre del módulo
"""
```

## 5. MANTENIMIENTO Y EVALUACIÓN

Ejemplo No 2

Enunciado

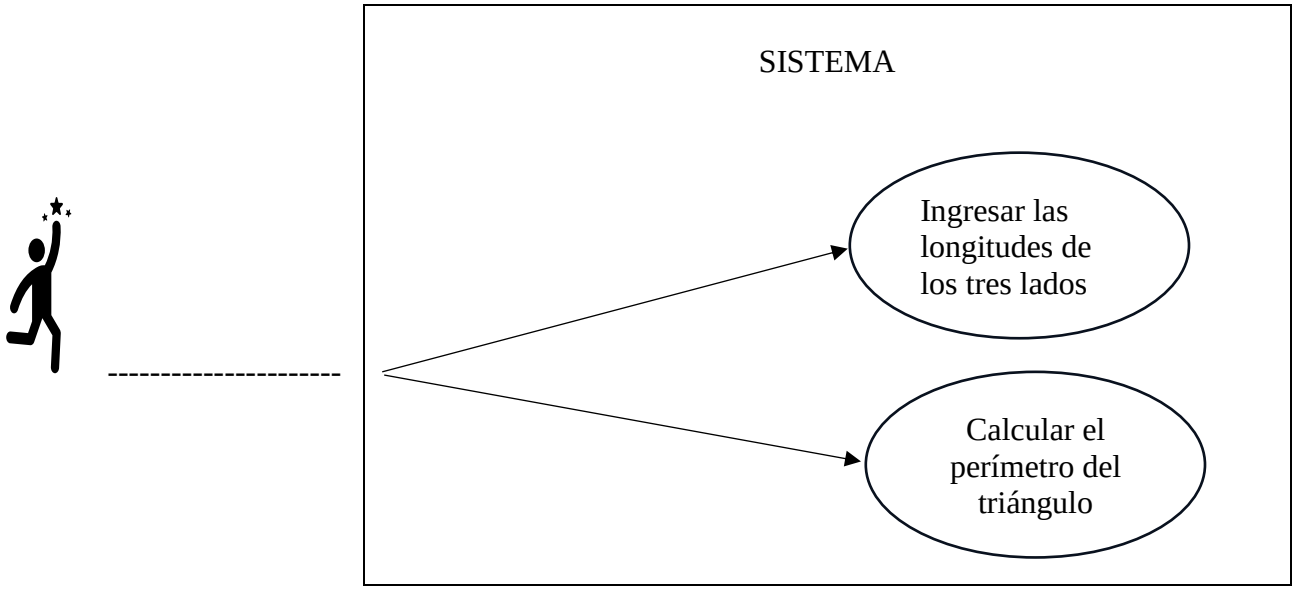
Calcular el perímetro de un triángulo cualquiera, dadas las longitudes de los tres lados.

1. IDENTIFICACION Y DELIMITACION DEL PROBLEMA A RESOLVER
   1.1 Identificación
   El enunciado es corto y trivial. Su interpretación no admite ambigüedades.

   1.2 Delimitación
   El problema está bien delimitado. Sólo es calcular el perímetro.

2. ANALISIS DEL PROBLEMA
   (Representación de casos de uso e identificación de clases, usando UML)

CASOS DE USO



2.1 Identificación de las clases

| Perimetro |
| --- |
| lado1<br>lado2<br>lado3 |
| calcularPerimetro() |

p : Perimetro

2.1.1 Documentación de los miembros de la clase

lado1 : Longitud del primer lado. Tipo entero.
lado2 : Longitud del segundo lado. Tipo entero.
lado3 : Longitud del tercer lado. Tipo entero.
calcularPerimetro() : Método para sumar los tres lados
p : Instancia u objeto de la clase Perimetro

2.2 Herencia de clases y/o Polimorfismo

No se presenta herencia de clases. No hay polimorfism

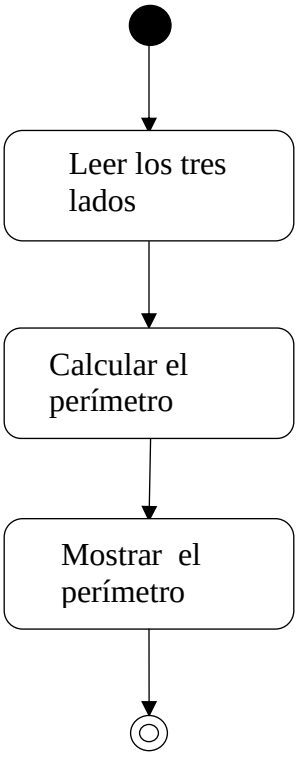
3. DESARROLLO DE LA SOLUCION

3.1 Modelación Matemática

Para este caso, se requiere un cálculo simple.
valorPerimetro = lado1 + lado2 + lado3

3.2 Propuesta de solución

3.2.1 Diagrama de actividades




3.2.2 Definición de las estructuras de datos

No existen.

3.2.3 Desarrollo del algoritmo (con diagramas N - S)

| Inicio | Operaciones |
|---|---|
| Declaración de clases, variables y constantes<br><br>Perímetro: p<br>lado1<br>lado2<br>lado3 | Perímetro: calculoPerimetro<br>retorne lado1 + lado2 + lado3 |

| Algoritmo principal |
|---|
| leer lado1<br>leer lado2<br>leer lado3<br>p = Perímetro(lado1, lado2, lado3)<br>valorPerimetro = p.calcularPerimetro()<br>imprimir valorPerimetro |

4. IMPLEMENTACIÓN

```
# -*- coding: utf-8 -*-
"""
Creado Octubre 27 de 2023
Nombre: manualPOOPython02
Que hace: Calcula el perímetro de un triángulo,
          dados los tres lados
Autores: Hernando Alvarez R
         Leonardo Alvarez V
         Luis Fernando Alvarez V
Universidad Tecnológica de Pereira

"""
class Perimetro:
    def __init__(self,lado1,lado2,lado3): # self representa la
        self.lado1 = lado1                # la instancia de la clase
        self.lado2 = lado2
        self.lado3 = lado3
    # Metódo de la clase
    def calculoPerimetro(self):
        return self.lado1+self.lado2+self.lado3

#Ejemplo de uso
def main():
    # Solicitar al usuario los tres lados
    lado1 = float(input("Entre lado 1:"))
    lado2 = float(input("Entre lado 2:"))
    lado3 = float(input("Entre lado 3:"))
    # Crear una instancia de la clase Perimetro
    p = Perimetro(lado1,lado2,lado3)
    # Obtener el perimetro
    valorPerimetro = p.calculoPerimetro()
    # Imprimir el perímetro
    print("Perimetro =",valorPerimetro)
if "__name__==__main__":
    main()

# En python no hay que declarar las variables.
# Las variables incluidas en una clase se denominan atributos.
```

5. MANTENIMIENTO Y EVALUACIÓN

Ejemplo No 3

Enunciado

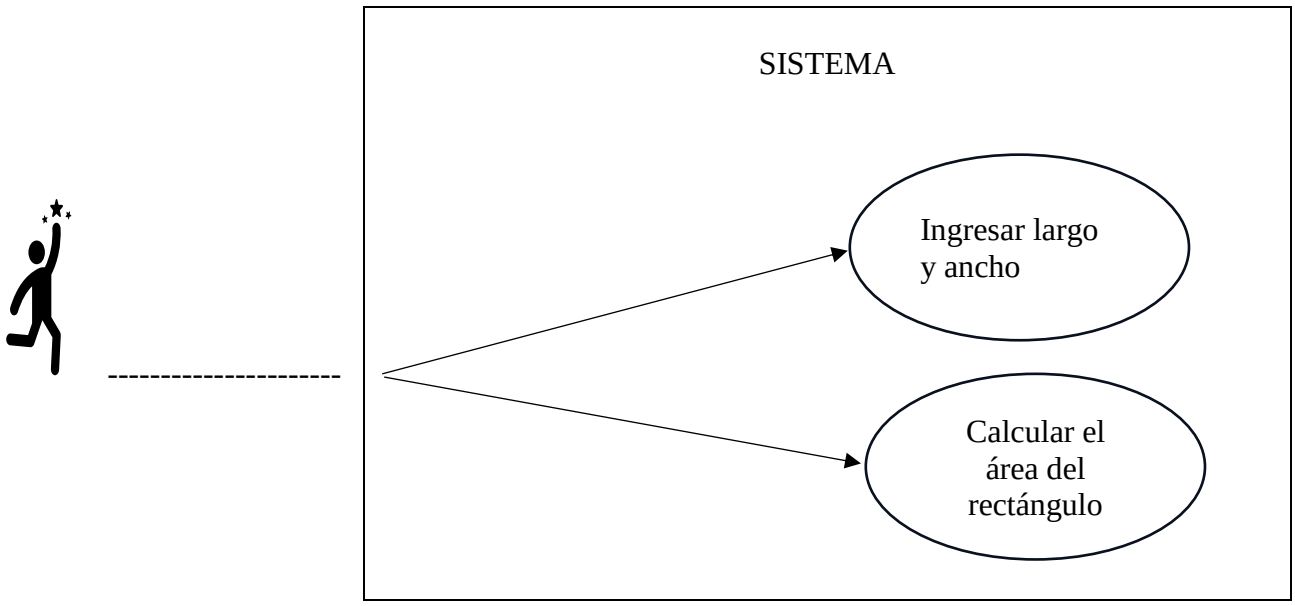
Dados el largo y el ancho de un rectángulo, calcular su área.

1. IDENTIFICACION Y DELIMITACION DEL PROBLEMA A RESOLVER
   1.1 Identificación
   El enunciado es corto y trivial. Su interpretación no admite ambigüedades.

   1.2 Delimitación
   El problema está bien delimitado. Sólo es calcular el área.

2. ANALISIS DEL PROBLEMA
   (Representación de casos de uso e identificación de clases, usando UML)

CASOS DE USO




2.1 Identificación de las clases

| Rectángulo |
| --- |
| largo<br>ancho |
| calcularArea() |

x : Rectangulo

2.1.1 Documentación de los miembros de la clase

largo : Largo del rectángulo. Tipo real o punto flotante
ancho : Ancho del rectángulo. Tipo real o punto flotante
CalcularArea() : Método para calcular el área del rectángulo
x : Instancia u objeto de la clase Rectangulo

2.2 Herencia de clases y/o Polimorfismo

No se presenta herencia de clases.
No hay polimorfismo.

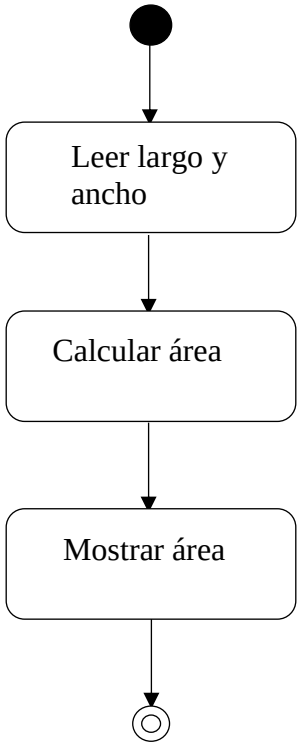
3. DESARROLLO DE LA SOLUCION

3.1 Modelación Matemática

Para este caso, se requiere un cálculo simple.
valorArea = largo * ancho

3.2 Propuesta de solución

3.2.1 Diagrama de actividades



3.2.2 Definición de estructuras de datos

No existen.

3.2.3Desarrollo del algoritmo (con diagramas N - S)

| Inicio | Operaciones |
|---|---|
| Declaración de clases, variables y constantes<br><br>Rectángulo: x<br>largo<br>ancho | Rectángulo :: calcularArea<br>retorne largo * ancho |
| Algoritmo principal | |
| leer largo<br>leer ancho<br>x = Rectangulo(largo, ancho)<br>valorArea = x.calcularArea ()<br>imprimir valorArea | |

4. IMPLEMENTACIÓN

```
# -*- coding: utf-8 -*-
"""
Creado Agosto 30 de 2023
Nombre: manualPOOPython03
Que hace: Calcula el área de un rectángulo si
          conocemos largo y ancho
Autores: Hernando Alvarez R.
         Leonardo Alvarez V.
         Luis Fernando Alvarez V.
Universidad Tecnológica de Pereira

"""
# Ejemplo de uso
class Rectangulo:
    def __init__(self, largo, ancho):
        self.largo = largo
        self.ancho = ancho

    def calcularArea(self):
        return self.largo * self.ancho

def main():
    # Solicitar largo y ancho
    largo = float(input("Entre largo: "))
    ancho = float(input("Entre ancho: "))
    # Crear una instancia de la clase Rectangulo
    x = Rectangulo(largo, ancho)
```

```
    # Obtener el valor del área e imprimirla
    valorArea = x.calcularArea()
    print("El área del rectangulo es: ",valorArea)
if "__name__ == __main__":
    main()
```

## 5. MANTENIMIENTO Y EVALUACIÓN

Ejemplo No 4

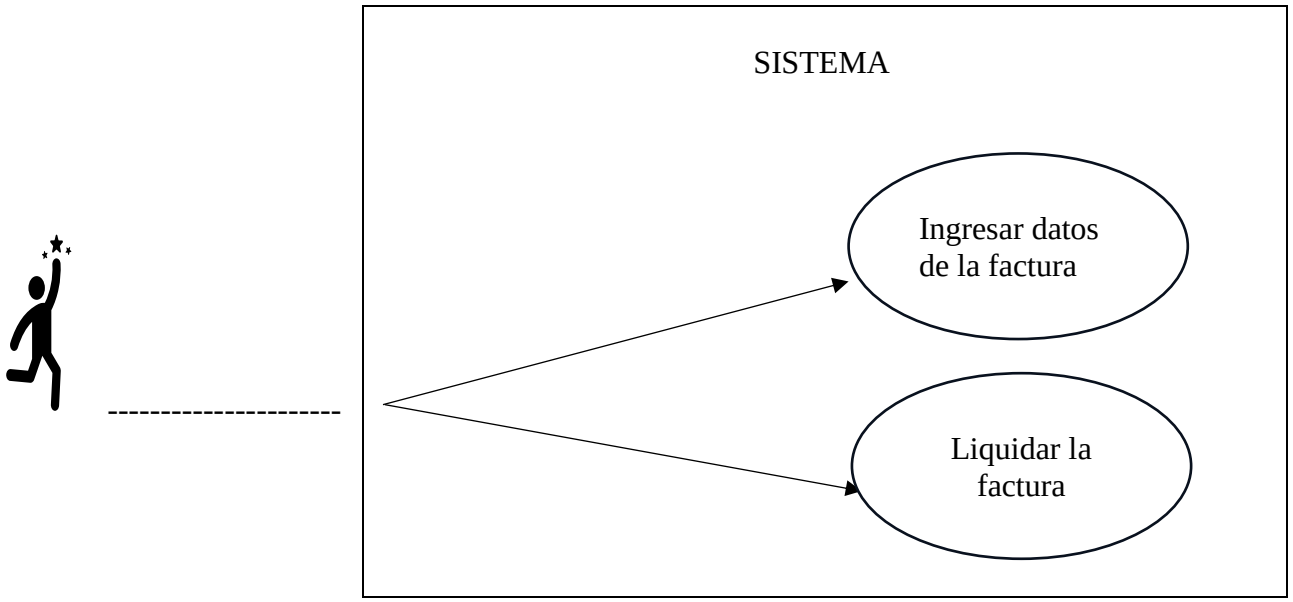
Enunciado

Liquidar una factura, sobre la cual se conoce:
código y nombre del artículo, número de artículos, costo de cada artículo y descuento en cada artículo.

1. IDENTIFICACION Y DELIMITACION DEL PROBLEMA A RESOLVER
   1.1 Identificación
   El enunciado es claro y no admite ambigüedades
   1.2 Delimitación
   El enunciado indica que se tiene una sola factura, con sus correspondientes datos.

2. ANALISIS DEL PROBLEMA
   (Representación de casos de uso e identificación de clases, usando UML)

CASOS DE USO



2.1 Identificación de las clase

| Factura | |
|---|---|
| descuento<br>codigo<br>articulo<br>narticulos<br>costoarticulo | x : Factura |
| leerDatos()<br>calcularFactura() | |

2.1.1 Documentación de los miembros de la clase

descuento : Descuento del artículo. Tipo real o punto flotante.
(Inicializar con el constructor de la clase)
codigo : Código del artículo. Tipo real o punto flotante.
articulo : Nombre del artículo. Tipo cadena.
narticulos : Número de artículos. Tipo entero
costoarticulo : Costo de cada artículo. Tipo real o punto flotante
leerDatos() : Método para leer la información
calcularFactura() : Método para calcular la factura
x : Instancia u objeto de la clase Factura

2.2 Herencia de clases y/o Polimorfismo
No se presenta herencia de clases.
No hay polimorfismo.

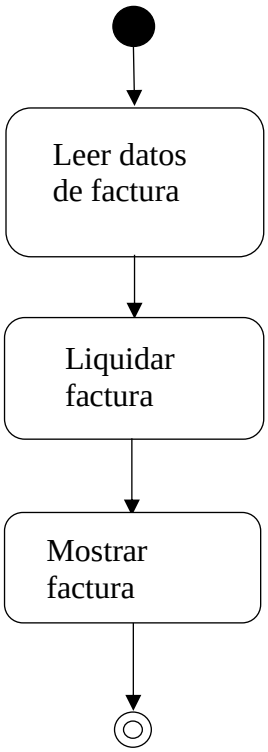
3. DESARROLLO DE LA SOLUCION

3.1 Modelación Matemática

Para este caso, se requiere el siguiente cálculo
valorFactura = narticulos * costoarticulo - descuento

3.2 Propuesta de solución

3.2.1 Diagrama de actividades




3.2.2 Definición de estructuras de datos

No existen en este programa.

3.2.3 Desarrollo del algoritmo (con diagramas N - S)

| Inicio | Operaciones |
|---|---|
| Declaración de clases, variables y constantes<br><br>Factura x<br>descuento<br>código = “ “<br>articulo = “ “<br>narticulos = 0<br>costoarticulo = 0 | Factura :: leerDatos<br>leer código, articulo,<br>narticulos, costoarticulo,<br><br>Factura :: calcularFactura<br>valorFactura = narticulos *<br>costoarticulo –<br>descuento<br>retorne valorFactura |

| Algoritmo principal |
| --- |
| x = Factura(50) # Inicializar el descuento<br>x.leerDatos()<br>netoFactura = x.calculaFactura()<br>imprimir netoFactura |

## 4. IMPLEMENTACIÓN

```
# -*- coding: utf-8 -*-
"""
Creado Octubre 25 de 2023
Nombre: manualPOOPython04
Que hace: Liquidar una factura
Autores: Hernando Alvarez R.
         Leonardo Alvarez V.
         Luis Fernando Alvarez V.
Universidad Tecnológica de Pereira
"""
class Factura: # Declaración de la clase
    def __init__(self,descuento):  # self representa la
        self.descuento = descuento # instancia de la clase
        self.codigo = ""
        self.articulo = ""
        self.narticulos = 0
        self.costoarticulo = 0
    def leerDatos(self): # Método de la clase
        # Datos de la factura
        self.codigo = input("Entre código del artículo:")
        self.articulo = input("Entre nombre del artículo:")
        self.narticulos = input("Entre # de artículos:")
        self.costoarticulo = input("Entre costo de cada artículo:")
        print("Código: ",self.codigo)
        print("Artículo: ",self.articulo)
    def calcularFactura(self): # Método para calcular valor factura
        valorFactura = int(self.narticulos)*int(self.costoarticulo)-
int(self.descuento)
        return(valorFactura)

# Ejemplo de uso
def main():
    x = Factura(50) # Inicializar el descuento
    x.leerDatos()
    netoFactura = x.calcularFactura()
    # Imprimir el valor de la factura
    print("El valor neto de la factura es:",netoFactura)
if '__name__ == __main__':
    main()

# En python no hay que declarar las variables
# Las variables incluidas en una clase se denominan atributos
```

# 5. MANTENIMIENTO Y EVALUACIÓN

Ejemplo No 5

Enunciado

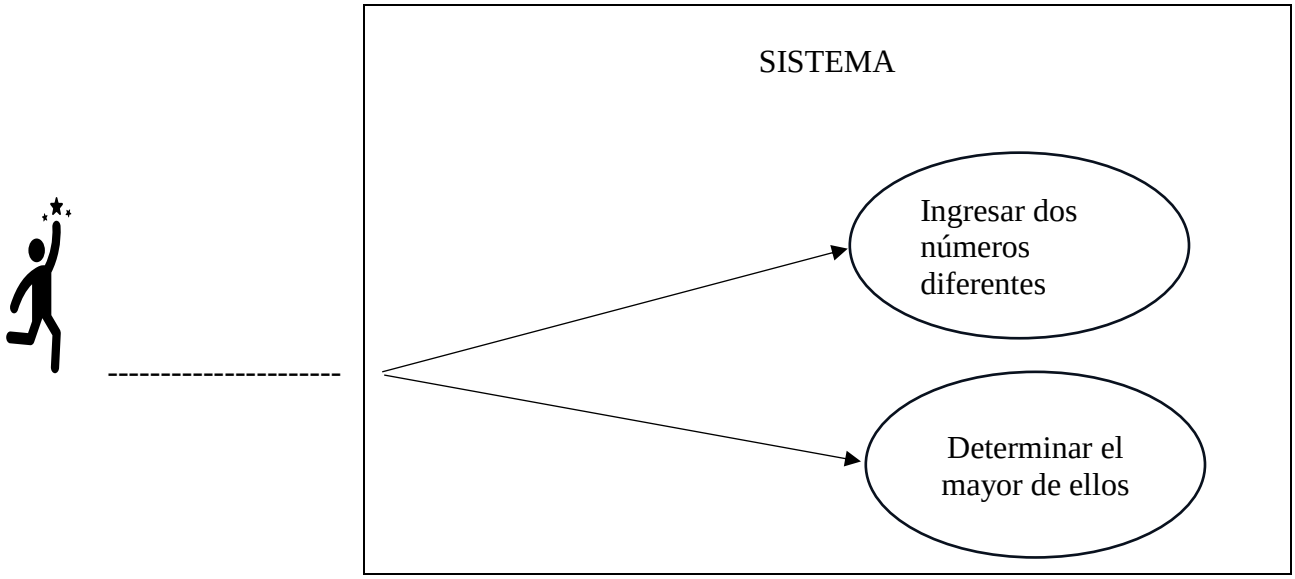
Leer dos números enteros y diferentes, para determinar cuál es el mayor de ellos

1. IDENTIFICACION Y DELIMITACION DEL PROBLEMA A RESOLVER
   1.2 Identificación
   Se trata de números enteros digitados por el usuario.

   1.2 Delimitación
   Los números deben ser enteros y diferentes.

2. ANALISIS DEL PROBLEMA
   (Representación de casos de uso e identificación de clases, usando UML)

CASOS DE USO



2.1 Identificación de las clases

| DeterminarMayor | |
|---|---|
| num1<br>num2 | comparador: DeterminarMayor |
| ingresar_numeros()<br>encontrar_mayor() | |

2.1.1 Documentación de los miembros de la clase

num1 : Primer número. Tipo entero
num2 : Segundo número. Tipo entero
ingresar_numeros() : Método para leer los dos números
encontrar_mayor() : Método para determinar el mayor de los dos números
comparador : Instancia u objeto de la clase DeterminarMayor

2.2 Herencia de clases y/o Polimorfismo

No se presenta herencia de clases.
No hay polimorfismo.

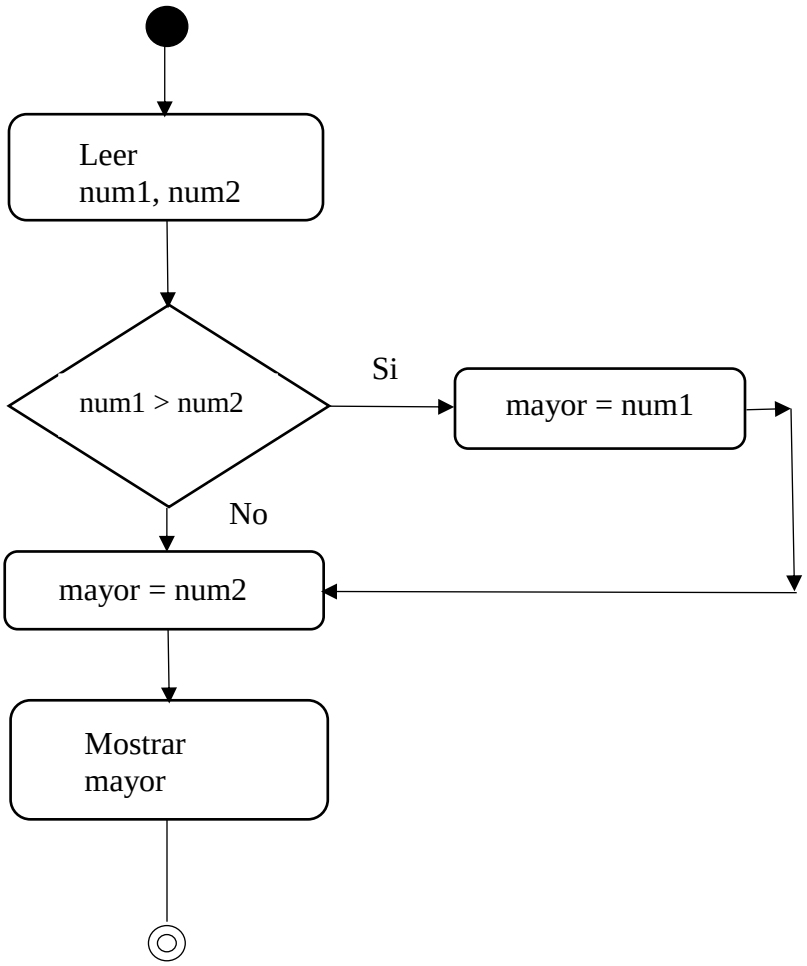
3. DESARROLLO DE LA SOLUCION

3.1 Modelación Matemática

Se requiere una comparación : Determinar el mayor de dos números enteros

3.2 Propuesta de solución

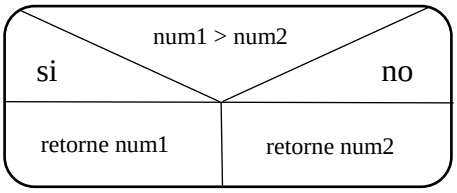
3.2.1 Diagrama de actividades




3.2.2 Definición de estructuras de datos

No existen.

3.2.3 Desarrollo del algoritmo (con diagramas N - S)

<table>
<tr><td>Inicio</td><td>Operaciones</td></tr>
<tr><td>Declaración de clases, variables y constantes<br><br>DeterminarMayor comparador<br><br>num1 = 0<br>num2 = 0</td><td>DeterminarMayor :: ingresar_numeros<br><br>leer num1<br>leer num2<br><br>DeterminarMayor :: encontrar_mayor<br> 
</td></tr>
<tr><td>Algoritmo principal</td><td></td></tr>
<tr><td>comparador = DeterminarMayor()<br>comparador.ingresar_numeros()<br>mayor = comparador.encontar_mayor()<br>imprimir (mayor)</td><td></td></tr>
</table>

## 4. IMPLEMENTACIÓN

```
# -*- coding: utf-8 -*-
"""
Creado Agosto 30 de 2023
Nombre: manualPOOPython05
Que hace: Dados dos números diferentes,
          determinar el mayor de ellos
Autores: Hernando Alvarez R
         Leonardo Alvarez V.
         Luis Fernando Alvarez V.
Universidad Tecnológica de Pereira

"""
class DeterminarMayor:
    def __init__(self): # self representa la
        self.num1 = 0   # instancia de la clase
        self.num2 = 0
    def ingresar_numeros(self):
        # Solicitar los dos números al usuario
        self.num1 = float(input("Ingrese primer número: "))
        self.num2 = float(input("Ingrese otro diferente al primero : "))
    def encontrar_mayor(self):
        # Determinar el número mayor
        if self.num1 > self.num2:
            return self.num1
        else:
```

```
            return self.num2

 # Ejemplo de uso
def main():
    # Crear una instancia de la clase DeterminarMayor
    comparador = DeterminarMayor()
    comparador.ingresar_numeros()
    mayor = comparador.encontrar_mayor()
    # Imprimir el número mayor
    print(f"El número mayor es: {mayor}")
if __name__ == "__main__":
    main()
```

## 5. MANTENIMIENTO Y EVALUACIÓN

Ejemplo No 6

Enunciado

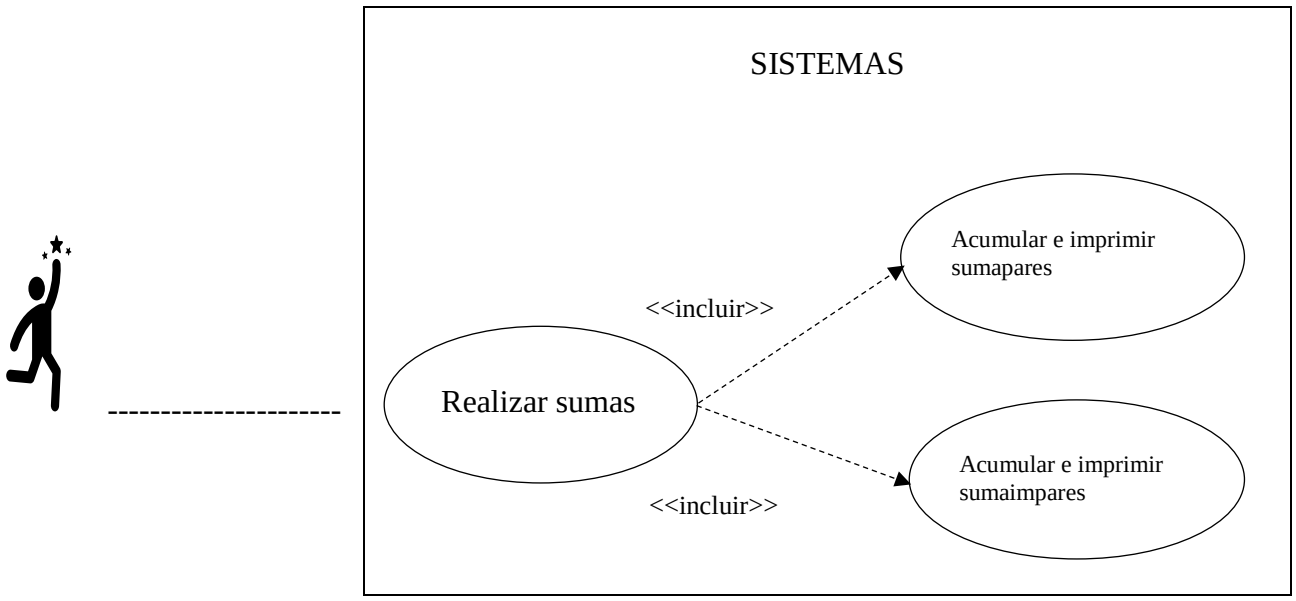
Calcular la suma de los números pares y la suma de los números impares comprendidos entre 2 y 10.
Se incluyen el 2 y el 10.

1. IDENTIFICACION Y DELIMITACION DEL PROBLEMA A RESOLVER
   1.1 Identificación
   1.2
   Calcular separadamente la suma de los números pares y de los números impares.

   1.2 Delimitación
   Se deben incluir el 2 y 10 en su correspondiente suma.

2. ANALISIS DEL PROBLEMA
   (Representación de casos de uso e identificación de clases)

CASOS DE USO



2.1 Identificación de las clases

| Pareseimpares | |
|---|---|
| n<br>suma_pares<br>suma_impares | x: Pareseimpares |
| calcular_sumas() | |

2.1.1 Documentación de los miembros de la clase

n : números (n=10) . Tipo entero
suma_pares : Suma de los números pares . Tipo entero
suma_impares : Suma de los números impares . Tipo entero
calcular_sumas() : Método para sumar números pares y números impares
x : Instancia u objeto de la clase Pareseimpares

2.2 Herencia de clases y/o Polimorfismo

No se presenta herencia de clases.
No hay polimorfismo.

3. DESARROLLO DE LA SOLUCION PARA CADA OPERACIÓN

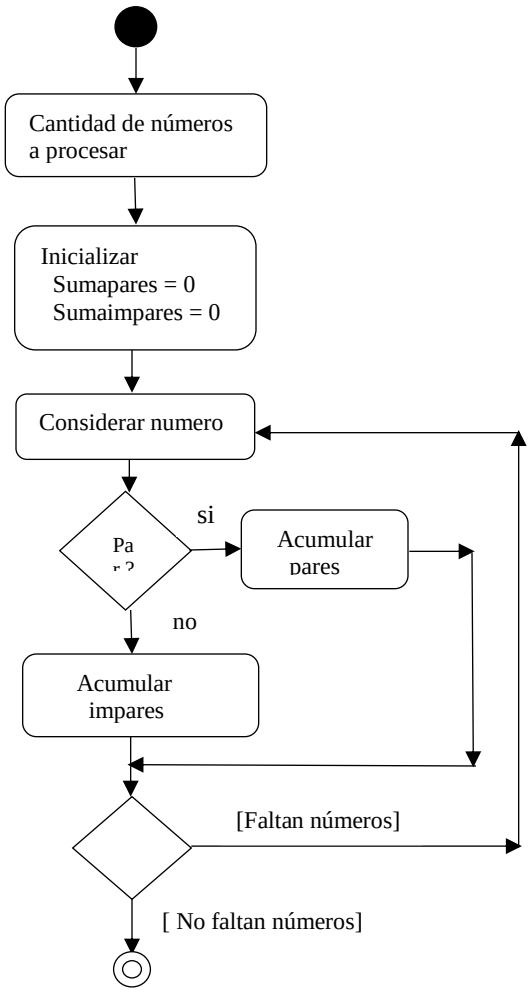
3.1 Modelación Matemática

Si i mod 2 = 0 entonces i es par

Si i mod 2 $\neq$ 0 entonces i es impar

3.2 Propuesta de solución

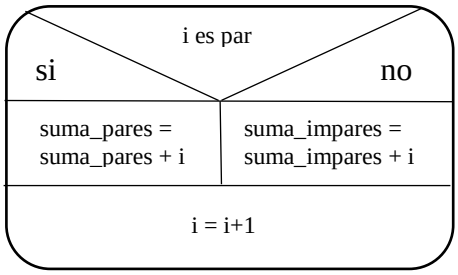
### 3.2.1 Diagrama de actividades




### 3.2.2 Definición de estructuras de datos

No existen.

### 3.2.3 Desarrollo del algoritmo (con diagramas N - S)

| Inicio | Operaciones |
|---|---|
| Declaración de clases, variables y constantes<br><br>Pareseimpares x<br>n<br>suma_pares = 0<br>suma_impares = 0 | Pareseimpares :: calcular_sumas<br>i ← 2<br>Mientras i < = n<br>  |
| Algoritmo principal | |
| x = Pareseimpares (10)<br>x . calcular_sumas () | |

## 4. IMPLEMENTACIÓN

```
# -*- coding: utf-8 -*-
"""
Creado Agosto 30 de 2023
Nombre: manualPOOPython06
Que hace: Calcula la suma de los números pares y
          los números impares, comprendidos entre 2 y 10,
          incluyendo el 2 y el 10
Autores: Hernando Alvarez R
         Leonardo Alvarez V
         Luis Fernando Alvarez V.
Universidad Tecnológica de Pereira
"""
class Pareseimpares:      # Clase principal
      def __init__(self,n): # self repsenta la
          self.n = n        # instancia de la clase
          self.suma_pares = 0
          self.suma_impares = 0
      def calcular_sumas(self): # Método principal
            i=2
            while i<=self.n:  # Ciclo mientras
                  if i%2==0: # Residúo de la división entera
                        self.suma_pares =self.suma_pares+i
                  else:
                        self.suma_impares = self.suma_impares+i
                  i=i+1 # Incremento
            print("Suma impares: ",self.suma_impares)
            print("Suma pares: ",self.suma_pares)
```

```
# Ejemplo de uso

def main(): # Programa principal
      x = Pareseimpares(10)
      x.calcular_sumas()
if '__name__ == __main__':
      main()
```

5. MANTENIMIENTO EVALUACION

Ejemplo No 7

Enunciado

Se tiene un total de n facturas y se desea calcular e imprimir al valor neto de cada una de ellas. Además, hallar el total por concepto de descuentos. Para cada factura se conoce su código, nombre del artículo, número de artículos y costo de cada artículo.

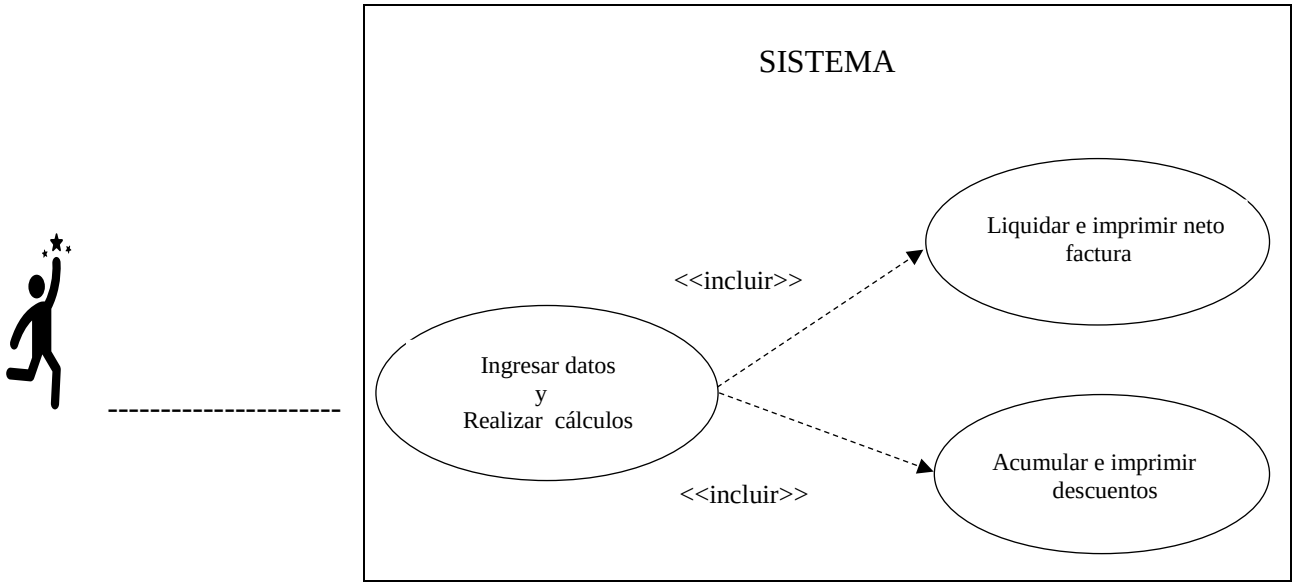
1. IDENTIFICACION Y DELIMITACION DEL PROBLEMA A RESOLVER

   1.1 Identificación
   El problema consiste en liquidar cada una de las facturas.

   1.2 Delimitación
   En cada fractura se tiene la información necesaria para realizar los cálculos.

2. ANALISIS DEL PROBLEMA
   (Representación de casos de uso e identificación de clases, usando UML)

CASOS DE USO



2.1 Identificación de las clases

| Facturas | |
|---|---|
| descuento<br>n<br>codigo<br>articulo<br>narticulos<br>costoarticulo<br>total_descuentos | factu: Facturas |
| leerDatos_y_liquidar() | |

2.1.1 Documentación de los miembros de la clase

descuento : Descuento en cada factura. Tipo real
n : Número de facturas. Tipo entero
codigo : código de cada artículo. Cadena de texto
articulo : Nombre de cada artículo. Cadena de texto
narticulos : Total de artículos. Tipo entero
total_descuento : Acumula los descuentos. Tipo real
leerDatos_y_liquidar : Método para leer datos y liquidar facturas
factu : Instancia u objeto de la clase Facturas

2.2 Herencia de clases y/o Polimorfismo

No se presenta herencia de clases.
No hay polimorfismo.

3. DESARROLLO DE LA SOLUCION PARA CADA OPERACIÓN

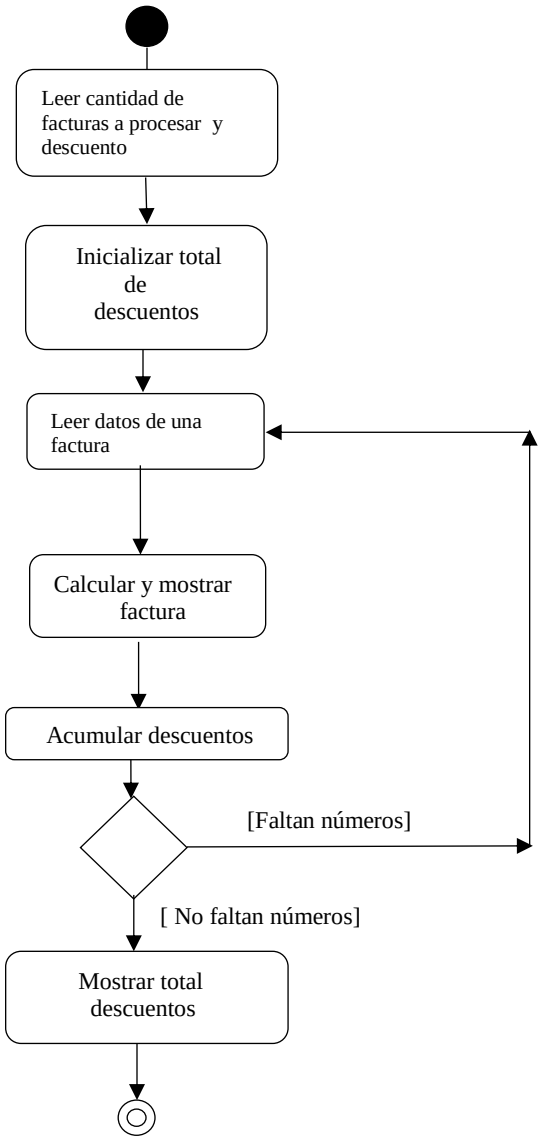
3.1 Modelación Matemática

liquida = narticulos * costo articulo - descuento

3.2 Propuesta de solución

3.2.1 Diagrama de actividades

Las entradas de datos y salida de información se resumen en el siguiente diagrama de actividades.




3.2.2 Definición de estructuras de datos. No existen.

### 3.2.4 Desarrollo del algoritmo (con diagramas N - S)

| Inicio | Operaciones |
|---|---|
| Declaración de clases, variables y constantes<br><br>Facturas factu<br>descuento<br>n<br>codigo = ‘‘ ’’<br>articulo = ‘‘ ’’<br>narticulo = 0<br>costoarticulo = 0<br>total_descuentos = 0 | Facturas :: leerDatos_y_liquidar<br>leer descuento, n<br>total_descuentos = 0<br>para j=1, n, 1<br><br>leer datos<br>imprimir datos<br>liquida = narticulos * costoarticulo – descuento<br>imprimir liquida<br>total_descuentos = total_descuentos + descuento<br><br>imprimir total_descuentos |
| Algoritmo principal | |
| factu = Facturas (5000,2)<br>factu.leerDatos_y_liquidar() | |

## 4. IMPLEMENTACION

```
# -*- coding: utf-8 -*-
"""
Creado Agosto 30 de 2023
Nombre: manualPOOPython07
Que hace: Liquidar n facturas
Autores: Hernando Alvarez R
         Leonardo Alvarez V
         Luis Fernando Alvarez V
Universidad Tecnológica de Pereira

"""

class Facturas: # Declaración de la clase
    def __init__(self, descuento, n):
        self.n = n
        self.descuento = descuento
        self.codigo = ""
        self.articulo = ""
        self.narticulos = 0
        self.costoarticulo = 0
        self.total_descuento = 0

    def leerDatos_y_liquidar(self): # Método de la clase
        # Datos de la facturas
        for j in [1, self.n]:
            self.codigo = input("Entre código del artículo:")
```

```
            self.articulo = input("Entre nombre del artículo:")
            self.narticulos = input("Entre # de artículos:")
            self.costoarticulo = input("Entre costo de cada artículo:")
            print("Código: ",self.codigo)
            print("Artículo: ",self.articulo)
            liquida=int(self.narticulos)*int(self.costoarticulo)-
                        int(self.descuento)
            print("Neto factura: ",liquida)
            self.total_descuento = self.total_descuento + self.descuento
        print("Total descuentos: ",self.total_descuento)

# Ejemplo de uso
def main(): # Programa principal
      factu = Facturas(5000,2)
      factu.leerDatos_y_liquidar()
if '__name__ == __main__':
      main()
```

5. MANTENIMIENTO Y EVALUACION

Ejemplo No 8

Enunciado

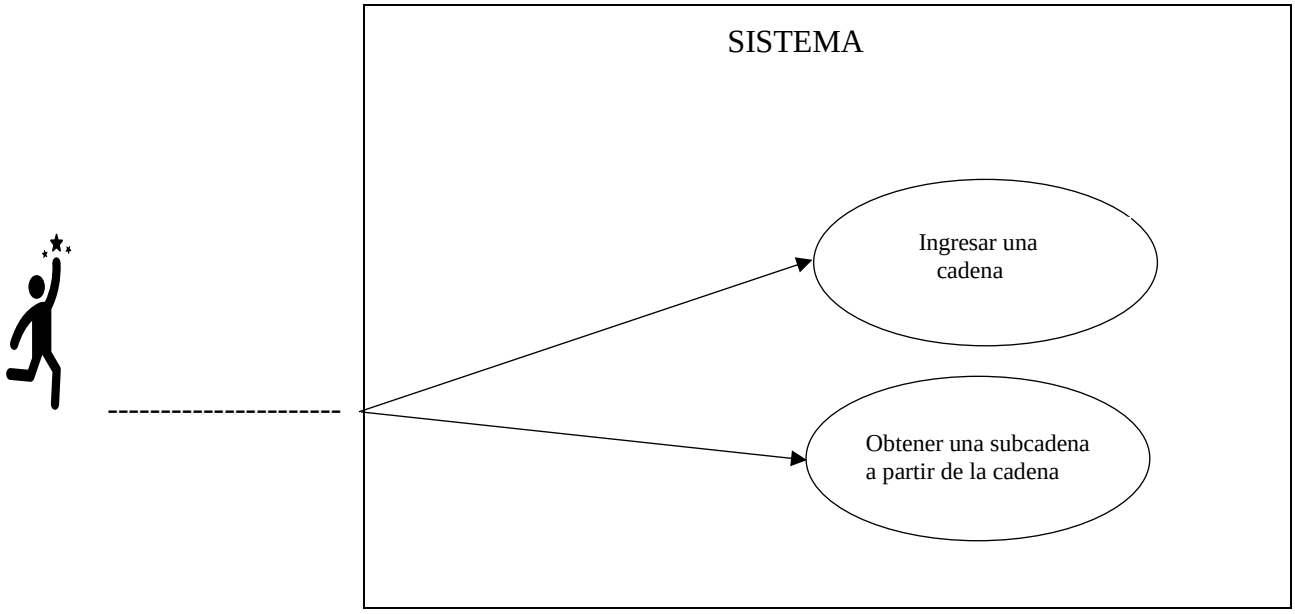
Dada una cadena, utilizar el concepto de rebanadas para obtener una subcadena, dando el inicio y el fin de la misma.

1. IDENTIFICACION Y DELIMITACION DEL PROBLEMA A RESOLVER
   1.1 Identificación
   Dada una cadena, a partir de esta obtener una subcadena.

   1.2 Delimitación
   Se debe indicar el comienzo y el final de la subcadena.

2. ANALISIS DEL PROBLEMA
   (Representación de casos de uso e identificación de clases)

CASOS DE USO



2.1 Identificación de las clases

| Rebanadas | |
|---|---|
| cadena | rebanar : Rebanadas |
| cortar_rebanada (inicio,fin) | |

2.1.1 Documentación de los miembros de la clase

cadena : Cadena para rebanar . Tipo cadena de caracteres

cortar_rebanada (inicio, fin) : Método para cortar rebanadas

s : Instancia u objeto de la clase Rebanadas

2.2 Herencia de clases y/o Polimorfismo

No se presenta herencia de clases.
No hay polimorfismo.

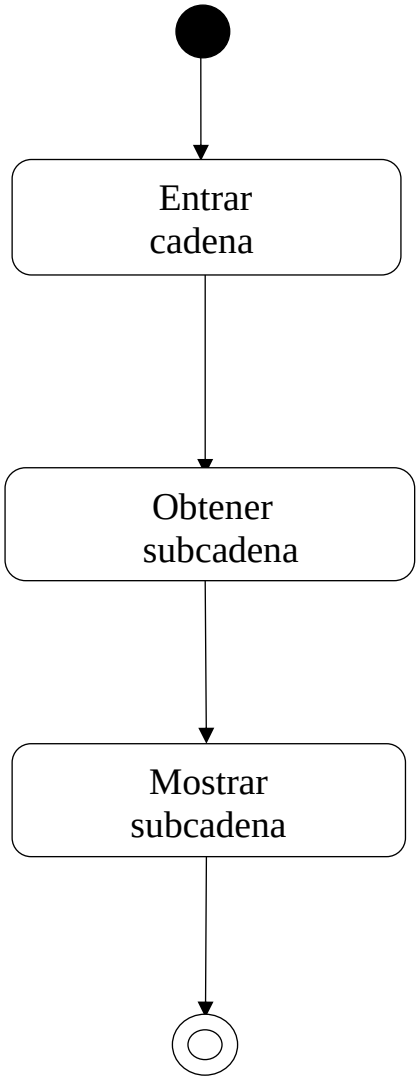
3. DESARROLLO DE LA SOLUCION PARA CADA OPERACIÓN

3.1 Modelación Matemática

No tenemos fórmulas matemáticas

3.2 Propuesta de solución

3.2.1 Diagrama de actividades




3.2.2 Definición de estructuras de datos

No existen.

3.2.3 Desarrollo del algoritmo (con diagramas N - S)

| Inicio | Operaciones |
|---|---|
| Declaración de clases, variables y constantes<br><br>Rebanadas rebanar<br>cadena | Rebanadas :: cortar_rebanas<br>retorne cadena[inicio : fin] |
| Algoritmo principal | |
| texto = “UNIVERSIDAD’’<br>rebanar = Rebanadas (texto)<br>imprimir rebanar . cortar_rebanadas (3,6) | |

## 4. IMPLEMENTACIÓN

```
# -*- coding: utf-8 -*-
"""
Creado Agosto 30 de 2023
Nombre: manualPOOPython08
Que hace: Se obtiene una rebanada a partir
          de una cadena
Autores: Hernando Alvarez R
         Leonardo Alvarez V
         Luis Fernando Alvarez V
Universidad Tecnológica de Pereira

"""
class Rebanadas: # Clase principal
    def __init__(self,cadena): # self representa
          self.cadena = cadena   # la instancia de la clase
    def cortar_rebanada(self, inicio, fin):
          return self.cadena[inicio:fin]

# Ejemplo de uso
def main(): # Programa principal
    texto = "UNIVERSIDAD"
    rebanar = Rebanadas(texto)
    # Los índices de las cadenas comienzan en 0
    # es decir: índice de U es 0, de N es 1, etc...
    # En rebanadas: [3:6] indica: índice 3 incluido
    # e índice 6 excluido
    print("Corte rebanada: ",rebanar.cortar_rebanada(3, 6))
if '__name__ == __main__':
      main()

# Las cadenas en python pueden estar encerradas en
# comillas dobles o sencillas
# Las cadenas de texto son inmutables
# El primer caracter de una cadena tiene índice 0
# Los índices negativos comienzan desde -1
# y este es el último caracter
```

```
# Las cadenas soportan rebanadas
```

5. MANTENIMIENTO EVALUACION

Ejemplo No 9

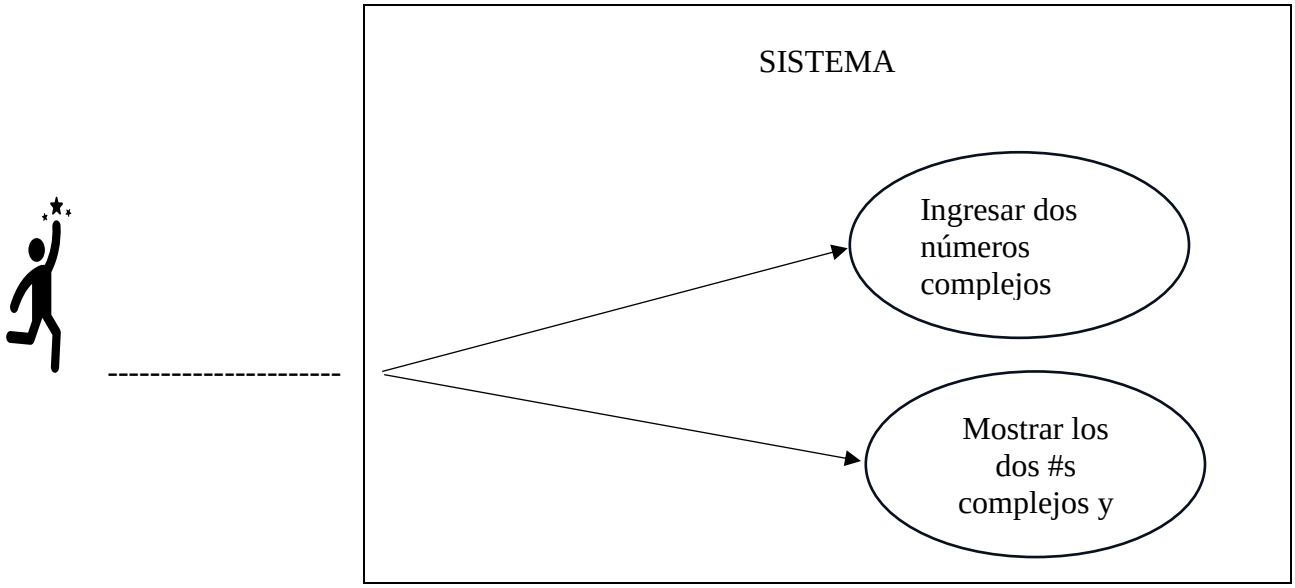
Enunciado

Dados dos números complejos, imprimirlos, calcular la suma, producto y división. Mostrar en pantalla los resultados.

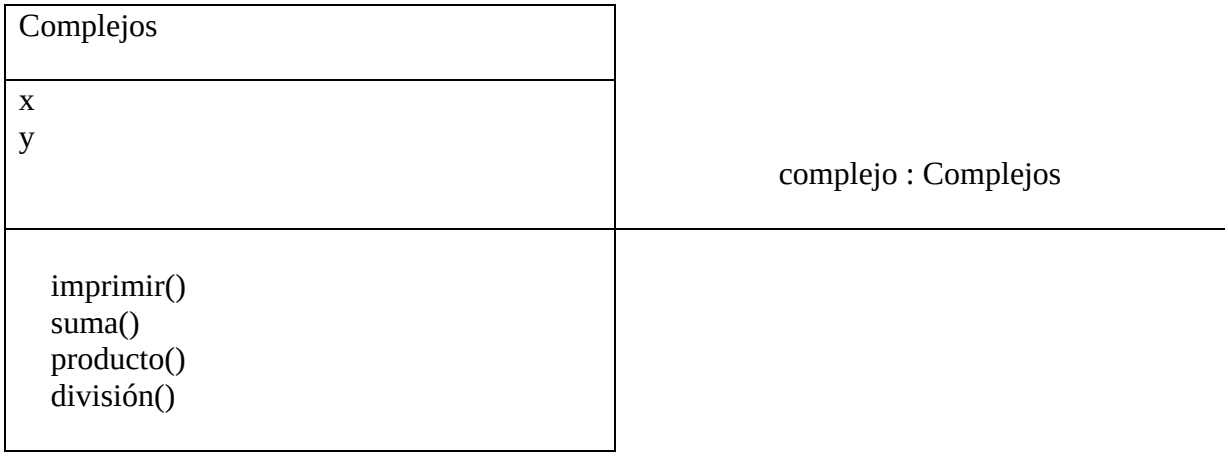
1. IDENTIFICACION Y DELIMITACION DEL PROBLEMA A RESOLVER
   1.1 Identificación
   La identificación no tiene problemas, pues solamente es leer los números complejos y realizar operaciones.

   1.2 Delimitación
   Se debe calcular e imprimir suma, producto y división de los dos números complejos.
2. ANALISIS DEL PROBLEMA
   (Representación de casos de uso e identificación de clases)

CASOS DE USO



2.1 Identificación de las clases




2.1.1 Documentación de los miembros de la clase

x : Primer número. Tipo complejo
y : Segundo número. Tipo complejo
imprimir() : Método para imprimir número complejo
suma() : Método para sumar números complejos
producto() : Método para multiplicar números complejos
división() : Método para dividir números complejos
complejo : Instancia u objeto de la clase Complejos

2.2 Herencia de clases y/o Polimorfismo

No se presenta herencia de clases.
No hay polimorfismo.

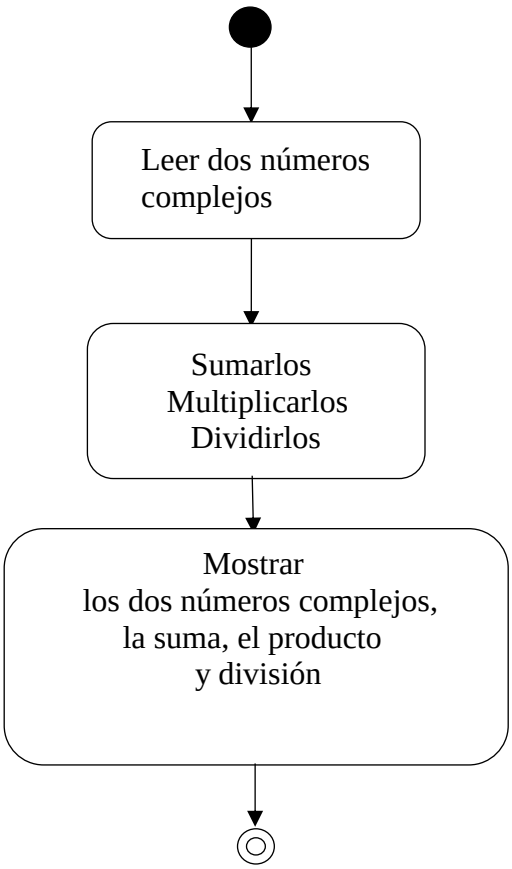
3. DESARROLLO DE LA SOLUCION

3.1 Modelación Matemática

Suma, producto y división de números complejos

3.2 Propuesta de solución

3.2.1 Diagrama de actividades




3.2.2 Definición de estructuras de datos

No existen.

3.2.3 Desarrollo del algoritmo (con diagramas N - S)

| Inicio | Operaciones |
|---|---|
| Declaración de clases, variables y constantes<br><br>Complejos complejo<br>x<br>y | Complejos :: imprimir<br>imprimir x<br>imprimir y<br><br>Complejos :: suma<br>retorne x+y<br><br>Complejos :: producto<br>retorne :: x*y<br><br>Complejos :: division<br>retorne :: x/y |
| Algoritmo principal | |
| complejo = Complejos (4+2j, 1+j)<br>complejo.imprimir<br>calcular e imprimir suma<br>calcular e imprimir producto<br>calcular e imprimir división | |

4. IMPLEMENTACIÓN

```
# -*- coding: utf-8 -*-
"""
Creado Agosto 30 de 2023
Nombre: manualPOOPython09
Que hace: Realizar operaciones con números complejos
Autor: Hernando Alvarez R
       Leonardo Alvarez V
       Luis Fernando Alvarez V.
Universidad Tecnológica de Pereira
"""
import math # Importartar la librería de Matemáticas
class Complejos: # Clase principal
    def __init__(self, x, y): # Parecido a un constructor
        self.x = x # self tiene la referencia del objeto
        self.y = y

    def imprimir(self):
        print(self.x) # Imprimir un número complejo
        print(self.y)

    def suma(self):# Suma de números complejos
        print("La suma es igual a: ",self.x+self.y)
```

```
    def producto(self):  # Producto de números complejos
        print("El producto es igual a: ",self.x*self.y)

    def division(self):  # División de números complejos
        print("La división es igual a: ",self.x/self.y)
# Ejemplo de uso
def main(): # Programa principal
    complejo = Complejos(4+2j,1+1j)
    complejo.imprimir()
    complejo.suma()
    complejo.producto()
    complejo.division()

if '__name__ == __main__':
      main()
```

5. MANTENIMIENTO Y EVALUACION

Ejemplo No 10

Enunciado

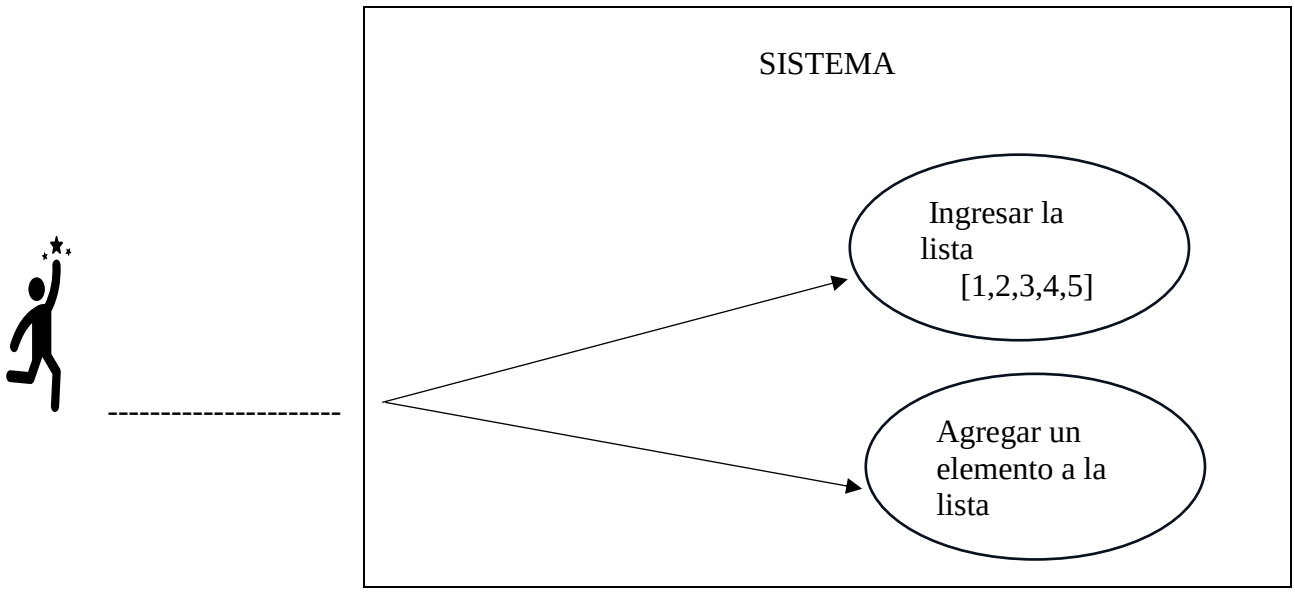
Agregar un elemento a lista [1,2,3,4,5] y luego imprimir todos sus elementos

1. IDENTIFICACION Y DELIMITACION DEL PROBLEMA A RESOLVER
   1.1 Identificación
   Dada una lista con cinco elementos, agregar un nuevo elemento a esta lista

   1.2 Delimitación
   Se debe usar la lista dada con sus cinco elementos

2. ANALISIS DEL PROBLEMA
   (Representación de casos de uso e identificación de clases)

CASOS DE USO



2.1 Identificación de las clases

| Manejalista | |
|---|---|
| lista | manejador_lista : Manejalista |
| agregar_elemento(elemento)<br>mostrar_lista() | |

2.1.1 Documentación de los miembros de la clase

lista : Lista de entrada. Tipo lista
agregar_elemento : Método para agregar un elemento a la lista
mostrar_lista() : Método para mostrar una lista
manejador_lista : Instancia u objeto de la clase Manejalista

2.2 Herencia de clases y/o Polimorfismo

No se presenta herencia de clases.
No hay polimorfismo.

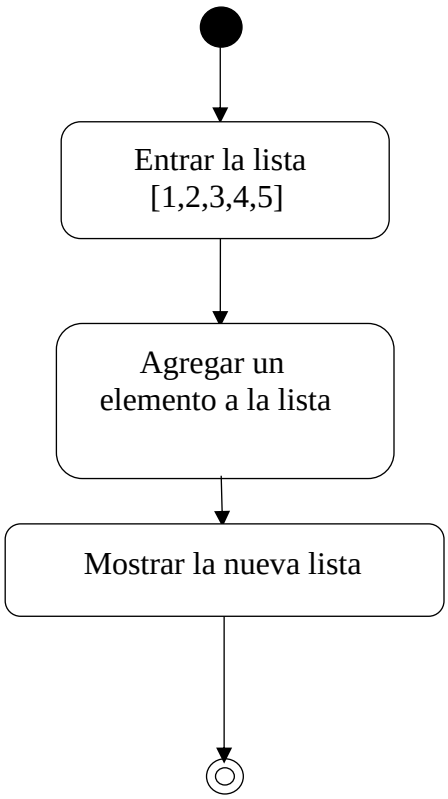
3. DESARROLLO DE LA SOLUCION PARA CADA OPERACION

3.1 Modelación Matemática

No se tienen fórmulas matemáticas. Solamente es agregar un elemento a la lista dada.

3.2 Propuesta de solución

3.2.1 Diagrama de actividades




3.2.2 Definición de estructuras de datos

lista: Tipo lista

3.2.3 Desarrollo del algoritmo (con diagramas N - S)

| Inicio | Operaciones |
|---|---|
| Declaración de clases, variables y constantes<br><br>Manejalista manejador_lista<br>lista | Manejalista :: agregar_elemento<br>agregar elemento<br><br>Manejalista :: mostrar_lista<br>para elemento en lista<br>imprimir elemento |
| Algoritmo principal | |
| mi_lista = [1,2,3,4,5]<br>manejador_lista = Manejalista (mi_lista)<br>leer elemento<br><br>manejador_lista.agregar_elemento(elemento)<br>manejador_lista.mostrar_lista() | |

4. IMPLEMENTACIÓN

```
# -*- coding: utf-8 -*-
"""
Creado Agosto 30 de 2023
Nombre: manualPOOPython10
Que hace: Crear un lista de números y luego agregar
          un elemento a esta lista
Autor: Hernando Alvarez R
       Leonardo Alvarez V
       Luis Fernando Alvarez V.
Universidad tecnológica de Pereira
"""
class Manejalista:
    def __init__(self, lista): # self representa
        self.lista = lista     # la instancia de la clase
    def agregar_elemento(self, elemento):
        self.lista.append(elemento)
        print(f"{elemento} ha sido agregado a la lista.")
    def mostrar_lista(self):
        print("Elementos en la lista:")
        for elemento in self.lista:
            print(elemento)
# Ejemplo de uso
def main(): # Programa principal
    # Lista inicial
```

```
    mi_lista = [1, 2, 3, 4, 5]
    # Crear una instancia del manejador de lista con la lista inicial
    manejador_lista = Manejalista(mi_lista)
    # Agregar un elemento a la lista
    elemento_a_agregar = int(input("Ingrese elemento a la lista: "))
    manejador_lista.agregar_elemento(elemento_a_agregar)
    # Mostrar todos los elementos de la lista
    manejador_lista.mostrar_lista()
if '__name__ == __main__':
      main()
# El tipo de datos lista es un tipo de datos compuesto
# Las listas son de tipo mutable
# Las listas pueden ser indexadas y soportan rebanadas
```

## 5. MANTENIMIENTO Y EVALUACION

Ejemplo No 11

Enunciado

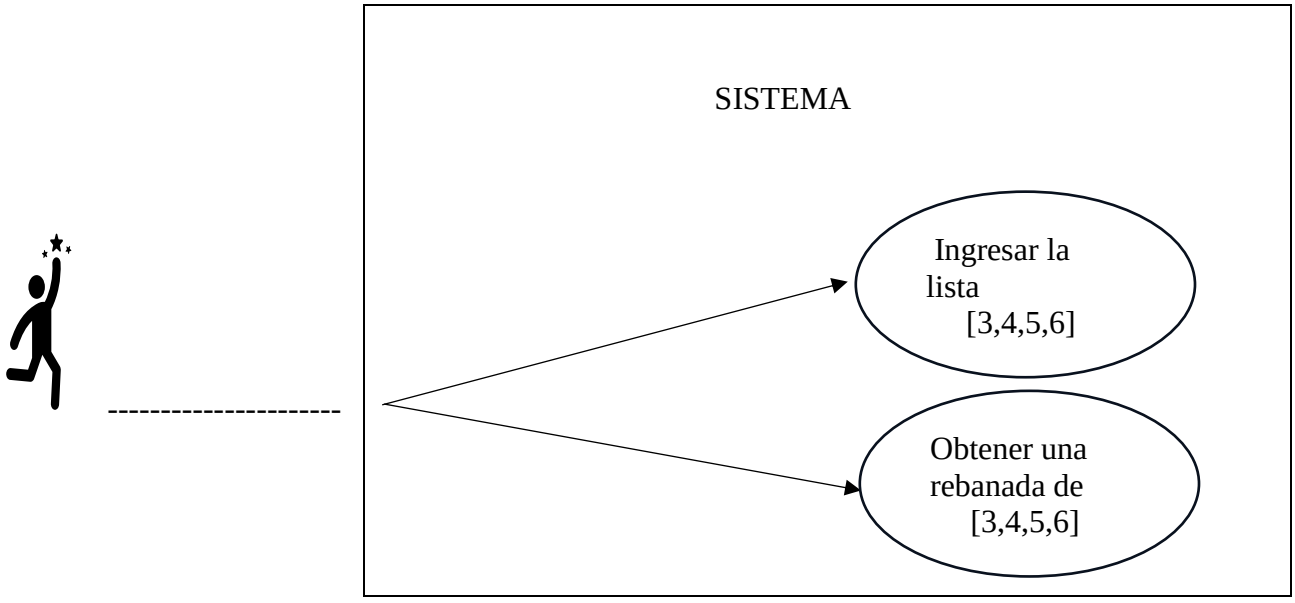
Crear una lista vacía y agregar los elementos 3,4,5 y 6 a la misma. Luego obtener una rebanada de esta última lista.

1. IDENTIFICACION Y DELIMITACION DEL PROBLEMA A RESOLVER
   1.1 Identificación
   Se trata de crear la lista [3,4,5,6] y luego a partir de esta, obtener una rebanada.

   1.2 Delimitación
   Se deben indicar el comienzo y el fin de la rebanada.

2. ANALISIS DEL PROBLEMA
   (Representación de casos de uso e identificación de clases, usando UML)

CASOS DE USO



2.1 Identificación de las clases

| CrearLista | |
|---|---|
| lista | manejador_lista : CrearLista |
| agregar_elemento(elemento)<br>mostrar_lista()<br>obtener_rebanada (inicio,fin) | |

2.1.1 Documentación de los miembros de la clase

lista : Lista de entrada. Tipo lista
agregar_elemento() : Método para agregar elemento a la lista
mostrar_lista() : Método para mostrar toda la lista
obtener_rebanada (inicio,fin) : Método para obtener rebanada
manejador_lista : Instancia u objeto de la clase CrearLista

2.2 Herencia de clases y/o Polimorfismo

No se presenta herencia de clases. No hay polimorfismo.

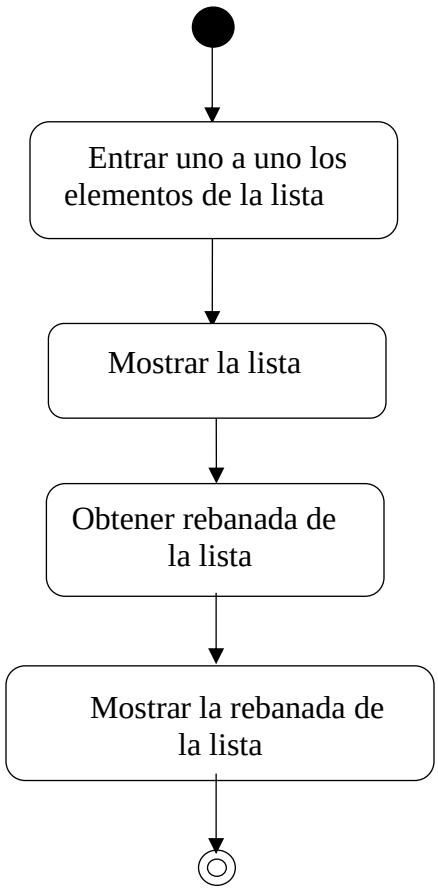
3. DESARROLLO DE LA SOLUCION PARA CADA OPERACION

3.1 Modelación Matemática

No se tienen fórmulas matemáticas. Se necesita el manejo de rebanadas en Listas

3.2 Propuesta de solución

3.2.1 Diagrama de actividades




3.2.2 Definición de estructuras de datos

rebanada: lista

3.2.3 Desarrollo del algoritmo (con diagramas N - S)

| Inicio | Operaciones |
|---|---|
| Declaración de clases, variables y constantes<br><br>CrearLista manejador_lista<br>lista = [] | CrearLista :: agregar_elemento<br>agregar elemento<br><br>CrearLista :: mostrar_lista<br>imprimir lista |
| Algoritmo principal | |
| manejador_lista=CrearLista()<br>manejador_lista.agregar_elemento(3)<br>manejador_lista.agregar_elemento (4)<br>manejador_lista.agregar_elemento (5)<br>manejador_lista.agregar_elemento (6)<br>manejador_lista.mostrar_lista()<br><br>inicio=1<br>fin=3<br>rebanada=manejador_lista.obtener_rebanada(inicio,fin)<br>imprimir rebanada | |

## 4. IMPLEMENTACIÓN

```
# -*- coding: utf-8 -*-
"""
Creado Agosto 30 de 2023
Nombre: manualPOOPython11
Que hace: Crear un lista vacía, agrega cuatro elementos y
          al final obtiene una rebanada
Autor: Hernando Alvarez R
       Leonardo Alvarez V
       Luis Fernando Alvarez V.
Universidad tecnológica de Pereira
"""
class CrearLista: # Clase principal
    def __init__(self):  # self representa la
        self.lista = []  # instancia de la clase
    def agregar_elemento(self, elemento):
        self.lista.append(elemento)
    def mostrar_lista(self):
        print("Lista actual:", self.lista)
    def obtener_rebanada(self, inicio, fin):
        if 0 <= inicio < len(self.lista) and 0 <= fin <= len(self.lista):
            return self.lista[inicio:fin]
        else:
            print("Error: Índices fuera de rango.")
            return None
```

```
# Ejemplo de uso
def main():
    manejador_lista = CrearLista()
    # Agregar elementos a la lista
    manejador_lista.agregar_elemento(3)
    manejador_lista.agregar_elemento(4)
    manejador_lista.agregar_elemento(5)
    manejador_lista.agregar_elemento(6)
    # Mostrar la lista
    manejador_lista.mostrar_lista()
    # Obtener rebanada de la lista
    inicio = 1
    fin = 3
    rebanada = manejador_lista.obtener_rebanada(inicio, fin)
    if rebanada is not None:
        print(f"Rebanada de índice {inicio} a índice {fin}: {rebanada}")
if __name__ == "__main__":
    main()
```

## 5. MANTENIMIENTO Y EVALUACION

Ejemplo No 12

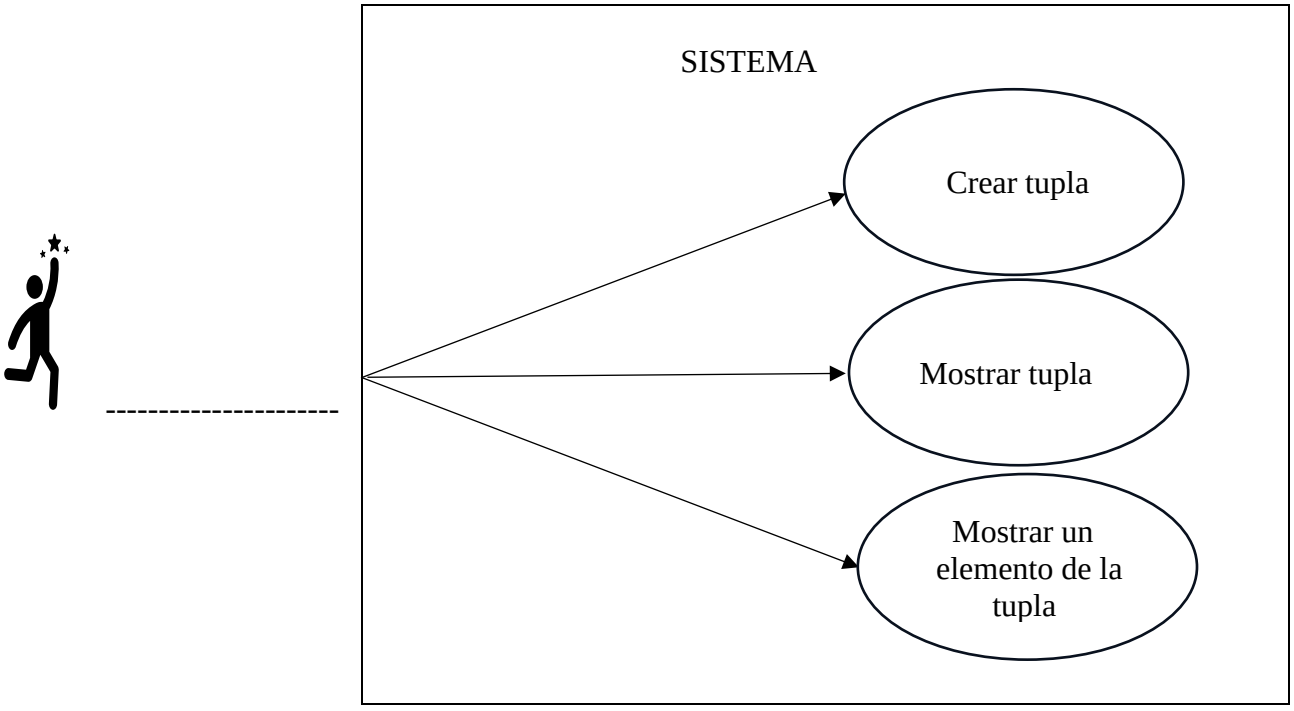
Enunciado

Crear una tupla vacía, agregar tres elementos e imprimir toda la tupla, luego obtener un elemento de la tupla dando su índice.

1. IDENTIFICACION Y DELIMITACION DEL PROBLEMA A RESOLVER
   1.1 Identificación
   Crear una tupla con tres elementos, mostrarla en pantalla y obtener un elemento de la misma.
   1.2 Delimitación
   La tupla debe tener tres elementos.

2. ANALISIS DEL PROBLEMA
   (Representación de casos de uso e identificación de clases, usando UML)

CASOS DE USO



2.1 Identificación de las clases

| ManejadorTupla | |
|---|---|
| mi_tupla | manejador : ManejadorTupla |
| agregar_elemento()<br>mostrar_tupla()<br>obtener_elemento () | |

2.1.1 Documentación de los miembros de la clase

mi_tupla : Tupla para adicionar elementos . Tipo tupla
agregar_elemento() : Método para agregar un elemento a la tupla
mostrar_tupla() : Método que muestra una tupla
obtener_elemento () : Método seleccionar un elemento de la tupla
manejador : Instancia u objeto de la clase ManejadorTupla

2.2 Herencia de clases y/o Polimorfismo

No se presenta herencia de clases.
No hay polimorfismo.

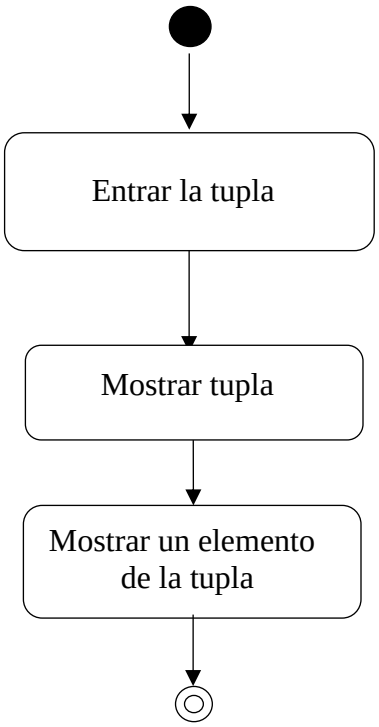
3. DESARROLLO DE LA SOLUCION PARA CADA OPERACION

3.1 Modelación Matemática

No se tienen fórmulas matemáticas.

## 3.2 Propuesta de solución

### 3.2.1 Diagrama de actividades




### 3.2.2 Definición de estructuras de datos

mi_tupla : tupla

3.2.3 Desarrollo del algoritmo (con diagramas N -S)

| Inicio | Operaciones |
|---|---|
| Declaración de clases, variables y constantes<br><br>ManejadorTupla manejador<br>mi_tupla. | ManejadorTupla :: agregar_elemento<br>agregar elemento<br><br>ManejadorTupla :: mostrar_tupla<br>imprimir mi_tupla<br><br>ManejadorTupla<br>::obtener_elemento(índice)<br>retorne mi_tupla (índice) |
| Algoritmo principal | |
| manejador=ManejadorTupla<br>manejador.agregar_elemento(7)<br>manejador.agregar_elemento(8)<br>manejador.agregar_elemanto(9)<br>manejador.mostrar_tupla()<br>índice=1<br><br>elemento=manejador.obtener_elemento(indice)<br>si elemento ≠ ∅<br>imprimir (elemento) | |

4. IMPLEMENTACIÓN

```
# -*- coding: utf-8 -*-
"""
Creado Agosto 30 de 2023
Nombre: manualPOOPython12
Que hace: Crear una tupla vacía, agregar tres elementos y
          luego mostrar toda la tupla
Autor: Hernando Alvarez R
       Leonardo Alvarez V
       Luis Fernando Alvarez V.
Universidad Tecnológica de Pereira
"""
class ManejadorTupla:
    def __init__(self): # self representa la
        self.mi_tupla = () # instancia de la clase
    def agregar_elemento(self, elemento):
        # Agrega un elemento a la tupla
        self.mi_tupla += (elemento,)
    def mostrar_tupla(self):
        # Muestra los elementos de la tupla
```

```
        print("Tupla actual:", self.mi_tupla)
    def obtener_elemento(self, indice):
        # Obtiene un elemento de la tupla dado un índice
        if 0 <= indice < len(self.mi_tupla):
            return self.mi_tupla[indice]
        else:
            print("Error: Índice fuera de rango.")
            return None

# Ejemplo de uso
def main():
    manejador = ManejadorTupla()
    # Agregar elementos a la tupla
    manejador.agregar_elemento(7)
    manejador.agregar_elemento(8)
    manejador.agregar_elemento(9)
    # Mostrar la tupla
    manejador.mostrar_tupla()
    # Obtener un elemento de la tupla
    indice = 1
    elemento = manejador.obtener_elemento(indice)
    if elemento is not None:
        print(f"Elemento en el índice {indice}: {elemento}")
if __name__ == "__main__":
    main()
# Las tuplas son inmutables
# La tupla realmente es una lista inmutable
# Las rebanadas funcionan en las tuplas
# como en las cadenas y listas
```

5. MANTENIMINETO Y EVALUACION

Ejemplo No 13

Enunciado

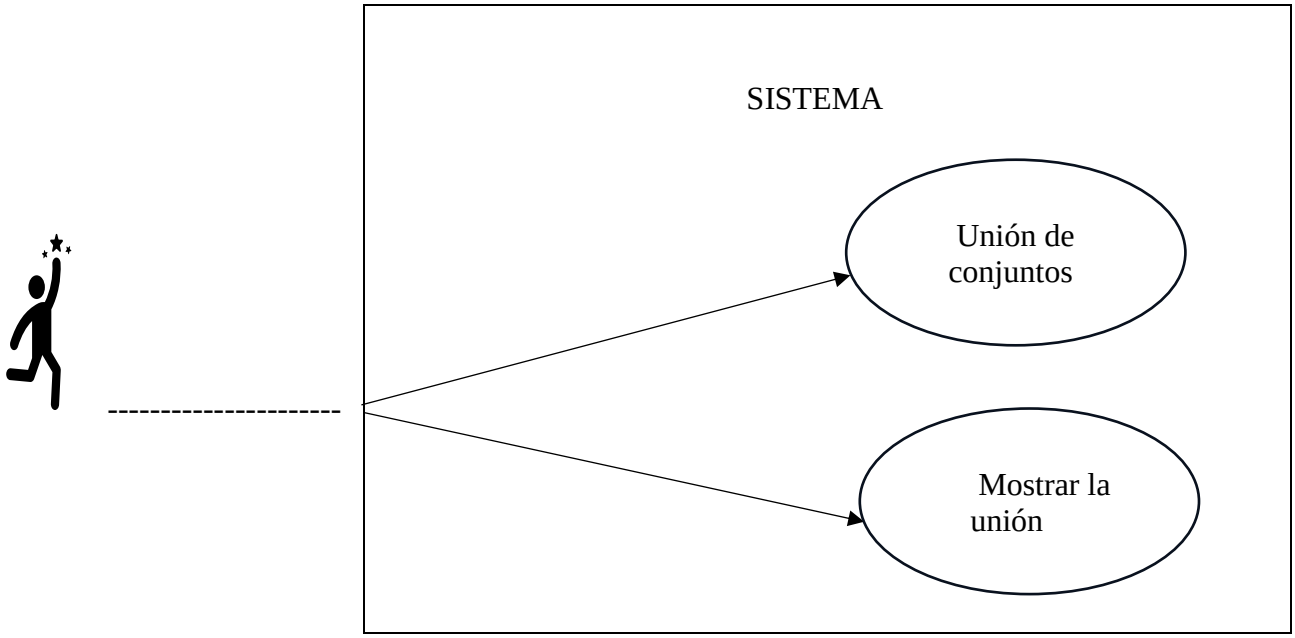
Dados dos conjuntos, cada uno con cuatro números enteros, hallar y mostrar su unión.

1. IDENTIFICACION Y DELIMITACION DEL PROBLEMA A RESOLVER
   1.1 Identificación
   Hallar la unión de dos conjuntos

   1.2 Delimitación
   Cada conjunto debe tener 4 números enteros

2. ANALISIS DEL PROBLEMA
   (Representación de casos de uso e identificación de clases, usando UML)

CASOS DE USO



2.1 Identificación de las clases

| Conjunto | |
|---|---|
| eles | conjunto1: Conjunto<br>conjunto2: Conjunto |
| union(otro_conjunto)<br>mostrar_conjunto() | |

2.1.1 Documentación de los miembros de la clase

Eles : lista vacía para convertir a conjunto. Tipo lista
union(otro_conjunto) : Método para realizar la unión de conjuntos
mostrar_conjunto() : Método para mostrar un conjunto
conjunto1 : Instancia(objeto) de la clase Conjunto
conjunto2 : Instancia(objeto) de la clase Conjunto

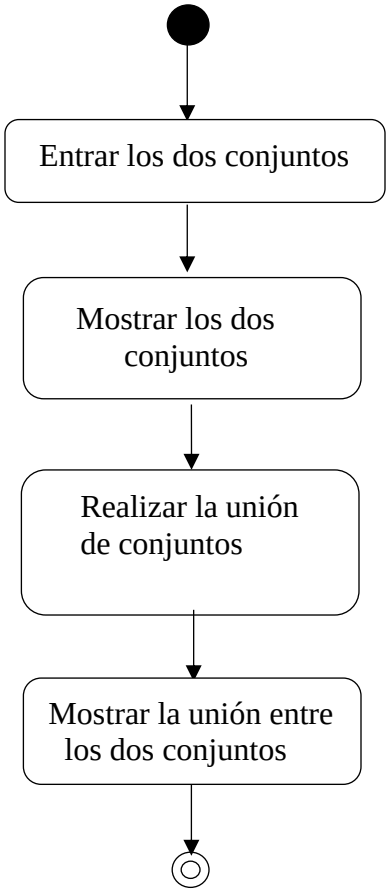
2.2 Herencia de clases y/o Polimorfismo

No se presenta herencia de clases.
No hay polimorfismo

3. DESARROLLO DE LA SOLUCION PARA CADA OPERACION

3.1 Modelación Matemática
No se tienen fórmulas matemáticas.
3.2 Propuesta de solución

3.2.1 Diagrama de actividades




3.2.2 Definición de estructuras de datos

No existen estructuras de datos.

3.2.4 Desarrollo del algoritmo (con diagramas N -S)

| Inicio | Operaciones |
|---|---|
| Conjunto conjunto1<br>Conjunto conjunto2<br>eles | Conjunto :: union(otro_conjunto)<br>nuevo_conjunto =<br>conjunto(eles.union(otro_conjunto.eles))<br>retorne nuevo_conjunto<br><br>Conjunto: mostrar_conjunto()<br>imprimir (eles) |
| Algoritmo principal | |
| conjunto1= Conjunto([3,4,5,6])<br>conjunto2= Conjunto([5,6,7,8])<br>conjunto1.mostrar_conjunto()<br>conjunto2.mostrar_conjunto()<br>union = conjunto1.union(conjunto2)<br>union.mostrar_conjunto() | |

4. IMPLEMENTACIÓN

```
# -*- coding: utf-8 -*-
"""
Creado Agosto 30 de 2023
Nombre: manualPOOPython13
Que hace: Unión entre conjuntos
Autores: Hernando Alvarez R
         Leonardo Alvarez V
         Luis Fernando Alvarez V.
Universidad tecnológica de Pereira
"""
class Conjunto: # Clase principal
                # elem: Se refiere a elemento
                # eles: Se refiere a elementos
    def __init__(self, eles=None): # self representa la
        if eles is None:           # instancia de la clase
            eles = []
        self.eles = set(eles)
    def union(self, otro_conjunto):
        nuevo_conjunto = Conjunto(self.eles.union(otro_conjunto.eles))
        return nuevo_conjunto
    def mostrar_conjunto(self):
        print(self.eles)

# Ejemplo de uso
def main():
    # Dos conjuntos para realizar la operación
    conjunto1 = Conjunto([3, 4, 5, 6])
```

```
    conjunto2 = Conjunto([5, 6, 7, 8])
    print("Conjunto 1:")
    conjunto1.mostrar_conjunto()
    print("\nConjunto 2:")
    conjunto2.mostrar_conjunto()
    # Operación unión
    union = conjunto1.union(conjunto2)
    print("\nUnión de Conjunto 1 y Conjunto 2:")
    union.mostrar_conjunto()
if __name__ == "__main__":
    main()
# En un conjunto no se utilizan índices
```

## 5. MANTENIMIENTO Y EVALUACION

Ejemplo No 14

Enunciado

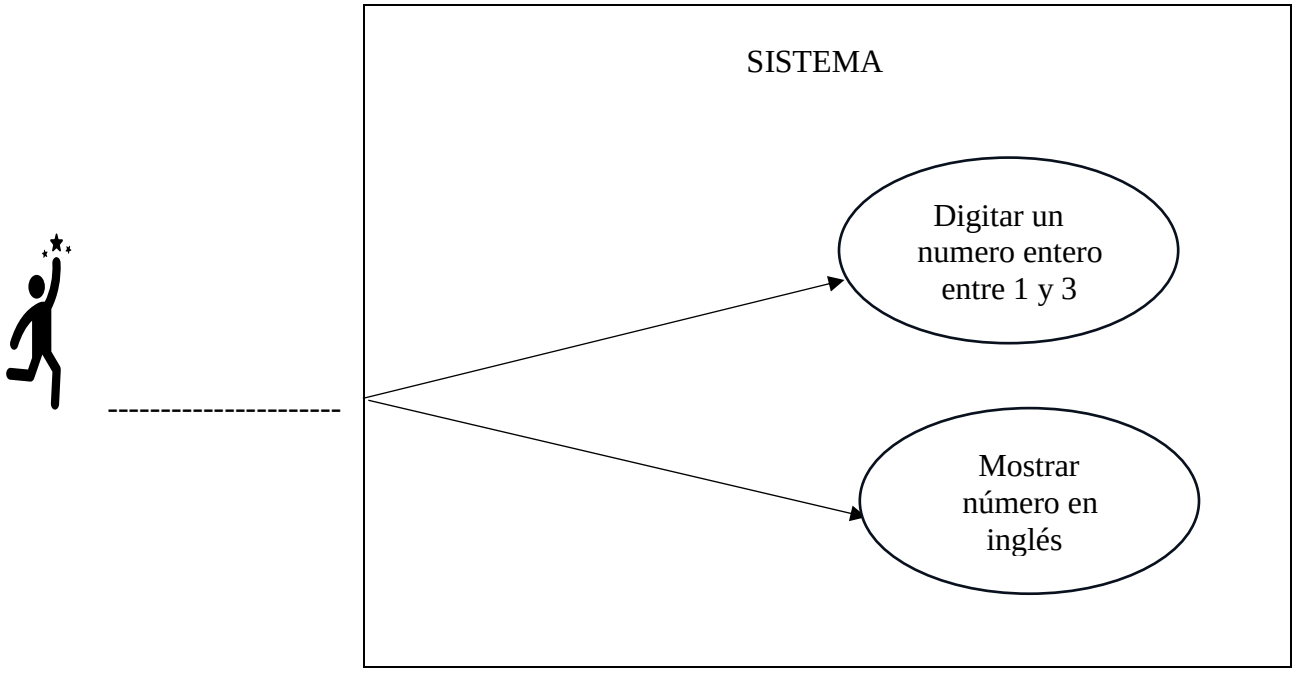
Traductor de inglés utilizando selección de casos.
Digitar un número entero comprendido entre el 1 y el 3, para luego escribirlo en inglés.

1. IDENTIFICACION Y DELIMITACION DEL PROBLEMA A RESOLVER
   1.1 Identificación
   Digitar número entero, para escribirlo en inglés.

   1.2 Delimitación
   El número digitado debe ser: 1,2 o 3

2. ANALISIS DEL PROBLEMA
   (Representación de casos de uso e identificación de clases, usando UML)

CASOS DE USO



2.1 Identificación de las clases

| Traductor | |
|---|---|
| num | traducir : Traductor |
| imprimir_ingles() | |

2.1.1 Documentación de los miembros de la clase
Num : Número a traducir .Tipo entero
imprimir_ingles() : Método para realizar la traducción
traducir : Instancia(objeto) de la clase Traductor

2.2 Herencia de clases y/o Polimorfismo

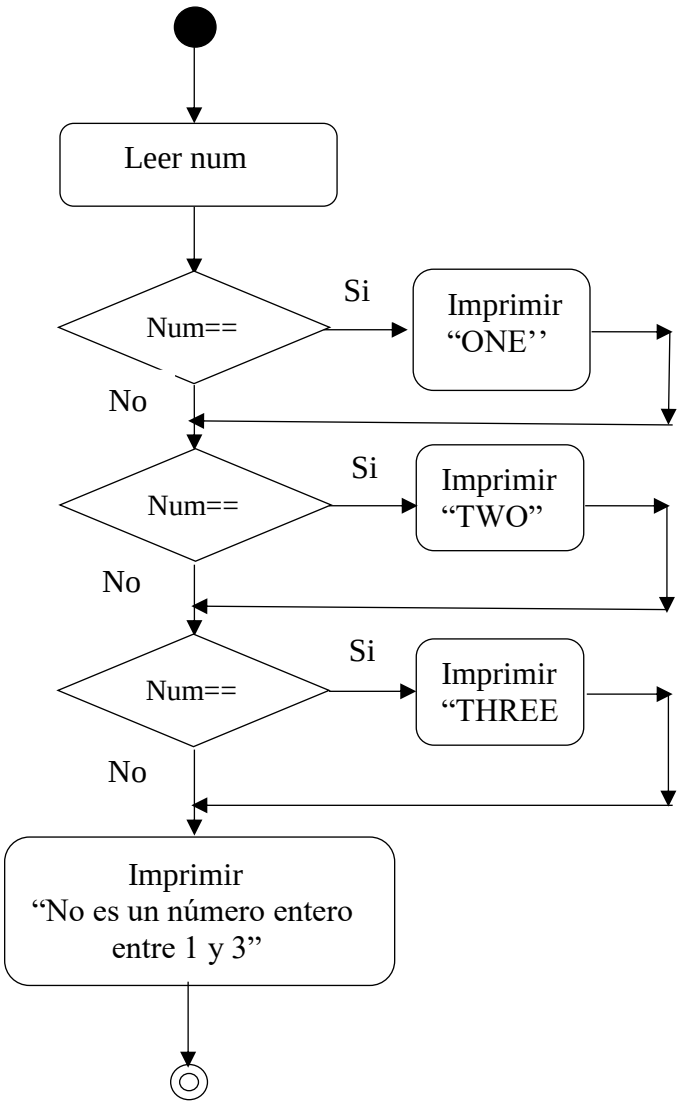
No se presenta herencia de clases.
No hay polimorfismo.

3. DESARROLLO DE LA SOLUCION PARA CADA OPERACION

3.1 Modelación Matemática
Se requiere una selección. No hay fórmula matemática

3.2 Propuesta de solución

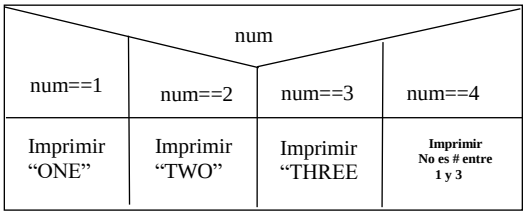
3.2.1 Diagrama de actividades




3.2.2 Definición de estructuras de datos

No existen estructuras de datos.

3.2.3 Desarrollo del algoritmo (con diagramas N -S)

| Inicio | Operaciones |
|---|---|
| Declaración de clases, variables y constantes<br><br>Traductor traducir<br>num | Traductor :: imprimir_ingles<br><br>leer num<br>  |
| Algoritmo principal | |
| num= Escriba ("entre un número de 1 a 3")<br>traducir= Traductor (num)<br>traducir.imprimir_ingles() | |

4. IMPLEMENTACION

```
# -*- coding: utf-8 -*-
"""
Creado Agosto 30 de 2023
Nombre: manualPOOPython14
Que hace: Traductor utilizando selección de casos
Autores: Hernando Alvarez R
         Leonardo Alvarez V
         Luis Fernando Alvarez V.
Universidad Tecnológica de Pereira
"""
class Traductor: # Clase principal
    def __init__(self, num):    # self representa la
        self.num = num           # instancia de la clase
    def imprimir_ingles(self):
        if self.num == 1:
            print ("ONE")
        elif self.num == 2:
            print ("TWO")
        elif self.num == 3:
            print ("THREE")
        else:
            print("No es un número entre 1 y 3")

# Ejemplo de uso
def main():
    # Solicitar un número al usuario
    num = int(input("Entre un número entre 1 y 3  _"))
    traducir = Traductor(num)
    traducir.imprimir_ingles()
```

```
if __name__ == "__main__":
    main()
```

5. MANTENIMIENTO Y EVALUACION.

Ejemplo No 15

Enunciado

Imprimir la sucesión de Fibonacci, hasta el número 13.

1. IDENTIFICACION Y DELIMITACION DEL PROBLEMA A RESOLVER

1.1 Identificación

Sucesión de Fibonacci hasta el número 13: 1,1,2,3,5,8,13: Primer elemento el 1; segundo elemento 1 y a partir del tercero, es la suma de los dos elementos anteriores.

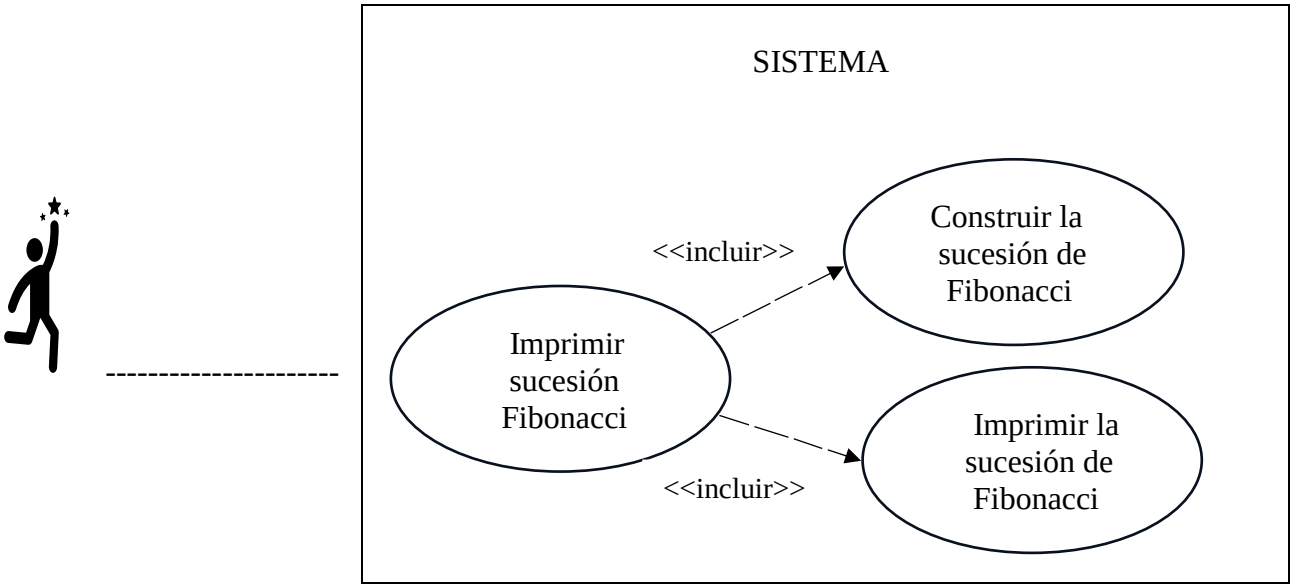
1.2 Delimitación

El último elemento de la sucesión es el número 13

2 ANALISIS DEL PROBLEMA

(Representación de casos de uso e identificación de clases, usando UML)

CASOS DE USO



2.1 Identificación de las clases

| Fibonacci |
| --- |
| L= []<br>n |
| Imprimir_fibo() |

fibo: Fibonacci

2.1.1 Documentación de los miembros de la clase

L : lista para almacenar los elementos de la lista. Tipo lista
n : guarda el número para generar la sucesión. Tipo entero
imprimir_fibo(): método para imprimir la sucesión de Fibonacci
fibo : Instancia(objeto) de la clase Fibonacci

2.2 Herencia de clases y/o Polimorfismo

No se presenta herencia de clases.
No hay polimorfismo.

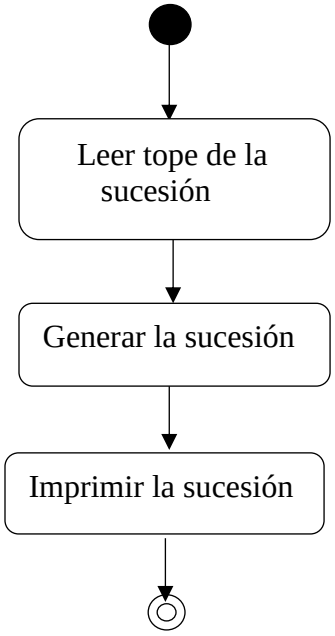
3 DESARROLLO DE LA SOLUCION PARA CADA OPERACION

3.1 Modelación Matemática

Se debe generar la sucesión de Fibonacci, hasta el número 13 así:
Dos primeros términos: el número 1
A partir de tercero, cada número es la suma de los dos anteriores.

3.2 Propuesta de solución

3.2.1 Diagrama de actividades




3.2.2 Definición de estructuras de datos

L: lista

3.2.3 Desarrollo del algoritmo (con diagramas N -S)

| Inicio | Operaciones |
|---|---|
| Declaración de clases, variables y constantes<br><br>Fibonacci fibo<br>L: [ ]<br>n | Fibonacci :: imprimir_fibo<br>a ◄— 0<br>b ◄— 1<br><br>mientras b < n<br>adicione b a L<br>a = b<br>b = a+b<br><br>para i en L<br>Imprimir (i) |
| Algoritmo principal | |
| fibo= Fibonacci(20)<br>fibo. imprimir_fibo() | |

4. IMPLEMENTACION

```
# -*- coding: utf-8 -*-
"""
Creado Agosto 30 de 2023

Nombre: manualPOOPython15
Que hace: Imprimir la sucesión de Fibonacci hasta
          el número 13
Autores: Hernando Alvarez R
         Leonardo Alvarez V
         Luis Fernando Alvarez V.
Universidad Tecnológica de Pereira
"""
class Fibonacci: # Clase principal
    L = []
    def __init__(self, n): # self representa la
        self.n = n         # instancia de la clase
    def imprimir_fibo(self):
        a, b = 0, 1
        while b < self.n:
            self.L.append(b)
            a, b = b, a + b
        for i in self.L:
            print(i)
```

```
# Ejemplo de uso
def main():
    fibo = Fibonacci(20)
    fibo.imprimir_fibo()
if __name__ == "__main__":
    main()
```

5.MANTENIMIENTO Y EVALUACION

Ejemplo No 16

Enunciado

Definir y utilizar un diccionario con tres pares clave: valor

1. IDENTIFICACION Y DELIMITACION DEL PROBLEMA A RESOLVER

1.1 Identificación

Se trata de un ejemplo sencillo, utilizando diccionarios.

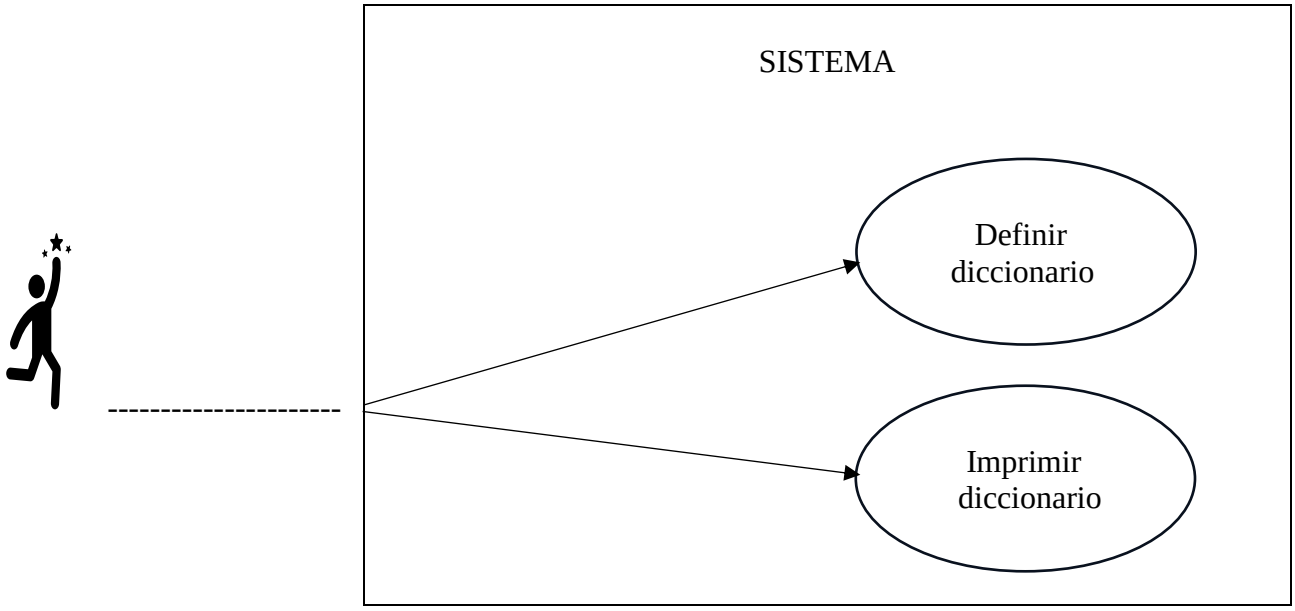
1.2 Delimitación

El diccionario está formado por tres pares clave:valor.

2 ANALISIS DEL PROBLEMA

(Representación de casos de uso e identificación de clases, usando UML)

CASOS DE USO



2.1 Identificación de las clases

| Diccionario | |
|---|---|
| undic | diccionario: Diccionario |
| imprimir_diccionario() | |

2.1.1 Documentación de los miembros de la clase

undic : Ejemplo de un diccionario. Tipo diccionario
imprimir_diccionario() : Método para imprimir diccionarios
diccionario: Instancia(objeto) de la clase Diccionario

2.2 Herencia de clases y/o Polimorfismo

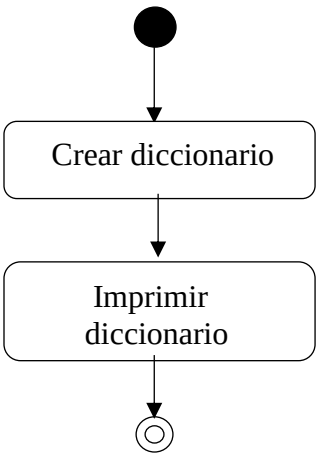
No se presenta herencia de clases.
No hay polimorfismo.

3. DESARROLLO DE LA SOLUCION PARA CADA OPERACION

3.1 Modelación Matemática
No hay fórmulas matemáticas

3.2 Propuesta de solución

3.2.1 Diagrama de actividades




3.2.2 Definición de estructuras de datos

undic: diccionario

3.2.4 Desarrollo del algoritmo (con diagramas N -S)

| Inicio | Operaciones |
|---|---|
| Declaración de clases, variables y constantes<br><br>UtilDiccionario  dicciona<br>undic: diccionario | UtilDiccionario :: imprimir_diccionario<br><br>imprimir undic<br>undic={}<br>imprimir undic<br>undic={'x':4,'y':-2,'z':1}<br>imprimir undic |
| Algoritmo principal | |
| dicciona= UtilDiccionario<br>dicciona.imprimir_diccionario() | |

Operaciones

4. IMPLEMENTACION

```
# -*- coding: utf-8 -*-
"""
Creado Agosto 30 de 2023
Nombre: manualPOOPython16
Que hace: Definir y utilizar un diccionario
Autores: Hernando Alvarez R
         Leonardo Alvarez V
         Luis Fernando Alvarez V.
Universidad Tecnológica de Pereira
"""
class UtilDiccionario: # Clase principal
    def __init__(self, undic = {}): # self representa la
        self.undic = undic          # instancia de la clase
    def imprimir_diccionario(self):
        print (self.undic)
        self.undic = {}
        print (self.undic)
        self.undic = {'x':4, 'y':-2, 'z':1}
        print(self.undic)

# Ejemplo de uso

def main():
    dicciona = UtilDiccionario({'Lunes':3, 'Martes':5, 'Miércoles':7})
    dicciona.imprimir_diccionario()
if __name__ == "__main__":
    main()

# Un diccionario es un conjunto ordenado de pares
# clave:valor, con la condición que las claves
# sean únicas
```

5.MANTENIMIENTO Y EVALUACION

Ejemplo No 17

Enunciado

Dada una lista de números enteros, utilizar la comprensión de listas para hallar otra lista solo con los números pares de la lista original.

1. IDENTIFICACION Y DELIMITACION DEL PROBLEMA A RESOLVER

1.1 Identificación

Dada una lista de números enteros, se debe crear otra lista solo con los números pares de la lista original

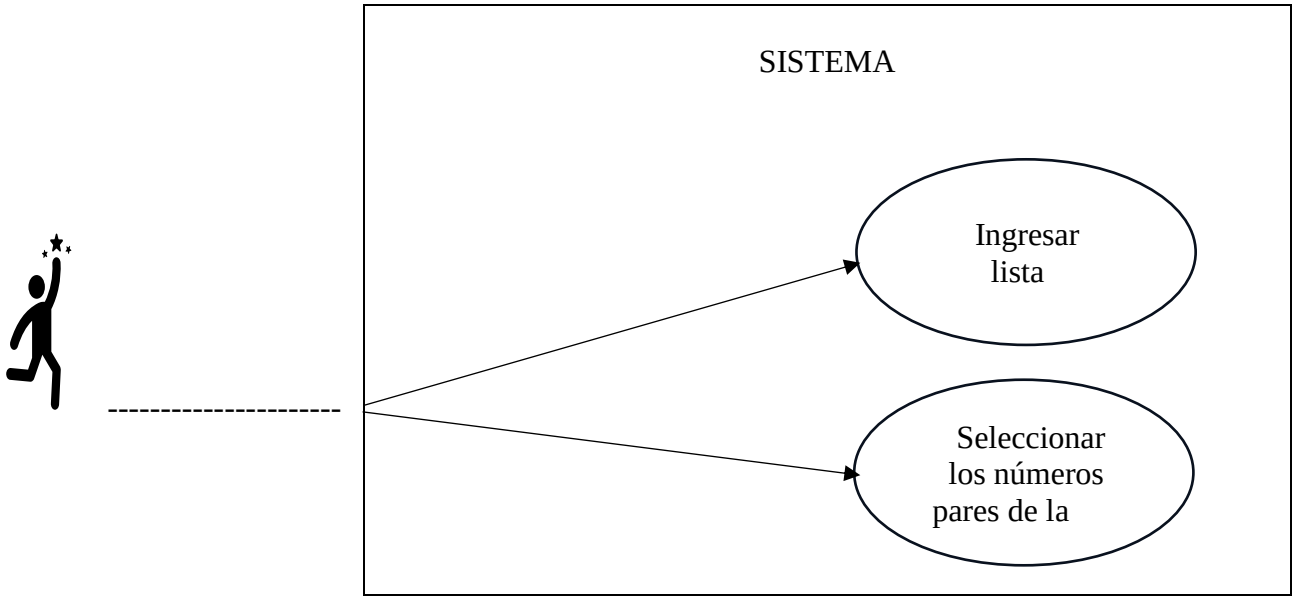
1.2 Delimitación

Obtener una nueva lista solo con los números pares.

4 ANALISIS DEL PROBLEMA

(Representación de casos de uso e identificación de clases, usando UML)

CASOS DE USO



2.1 Identificación de las clases

| ComprensionLista |
| --- |
| listaentrada<br><br>resultado: ComprensionLista |
| pares() |

2.1.1 Documentación de los miembros de la clase

listaentrada : lista de entrada. Tipo lista
pares() : Método para seleccionar pares
resultado : Instancia(objeto) de la clase ComprensionLista

2.2 Herencia de clases y/o Polimorfismo

No se presenta herencia de clases.
No hay polimorfismo.

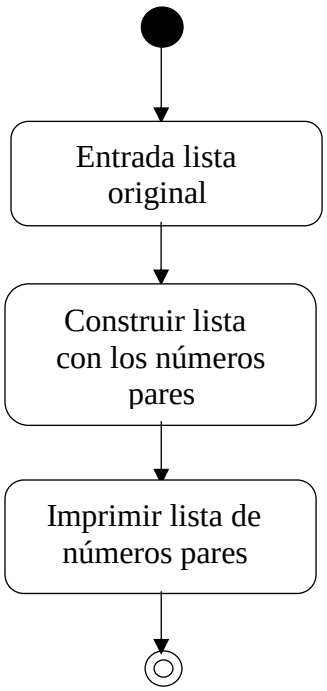
3 DESARROLLO DE LA SOLUCION PARA CADA OPERACION

3.1 Modelación Matemática

Si x % 2= 0, entonces x es par

3.2 Propuesta de solución

3.2.1 Diagrama de actividades




3.2.2 Definición de estructuras de datos

datos_entrada: lista

3.2.3 Desarrollo del algoritmo (con diagramas N -S)

| Inicio | Operaciones |
|---|---|
| Declaración de clases, variables y constantes<br><br>Comprensionlista resultado<br>listaentrada | ComprensionLista::pares<br><br>retorne [x para x en lista si x % 2==0] |
| Algoritmo principal | |
| datos_entrada=[3,4,5,6,7]<br>resultado=ComprensionLista(datos_entrada)<br>imprimir resultado.pares | |

## 4 IMPLEMENTACION

```
# -*- coding: utf-8 -*-
"""
Creado Agosto 30 de 2023
nombre: manualPOOPython17
Que hace: Muestra una operación, utilizando
          la comprensión de listas
Autores: Hernando Alvarez R
         Leonardo Alvarez V
         Luis Fernando Alvarez V.
Universidad Tecnológica de Pereira
"""
class ComprensionLista: # Clase principal
    def __init__(self, listaentrada): # self representa la instancia
        self.listaentrada = listaentrada     # de la clase
    # Seleccionar números pares de la lista
    def pares(self):
        return [x for x in self.listaentrada if x % 2 == 0]

# Ejemplo de uso

def main():
    datos_entrada = [3, 4, 5, 6, 7] # Datos de entrada
    resultado = ComprensionLista(datos_entrada)
    print("Números pares en la lista:", resultado.pares())
if __name__ == "__main__":
    main()


# La comprension de listas, es una forma adecuada
      # para crear listas
```

## 5.MANTENIMIENTO Y EVALUACION

Ejemplo No 18

Enunciado

Dada una tupla de parejas ordenadas, utilizar la comprensión de listas, para crear una lista con cada una de las sumas de componentes que forman cada pareja ordenada.

1. IDENTIFICACION Y DELIMITACION DEL PROBLEMA A RESOLVER

1.1 Identificación

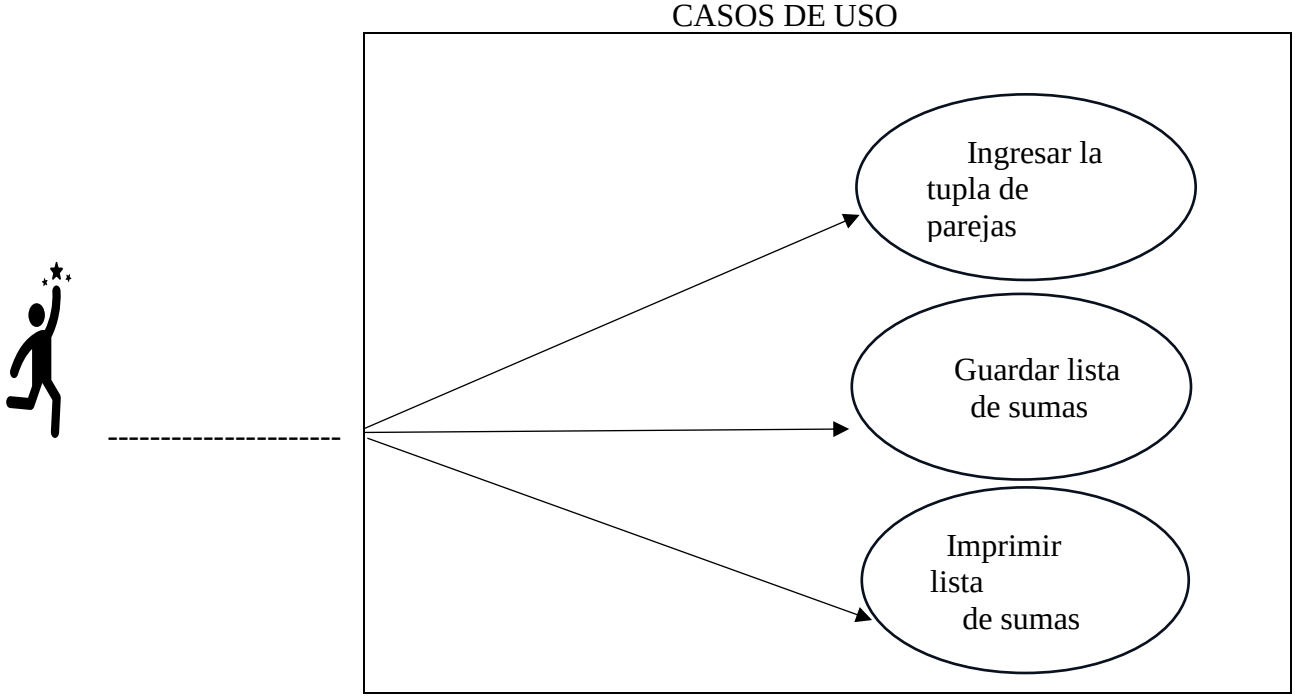
Cada elemento de la lista creada, está formado por la suma de las componentes de las parejas ordenadas.

1.2 Delimitación

Para generar la lista de sumas, se debe utilizar la comprensión de listas

2. ANALISIS DEL PROBLEMA

(Representación de casos de uso e identificación de clases, usando UML)



2.1 Identificación de las clases

| ComprensionParejas |
| --- |
| tupla |
| sumar_elementos() |

manejador_tupla: ComprensionParejas

2.1.1 Documentación de los miembros de la clase

tupla : tupla de parejas ordenadas. Tipo tupla
sumar_elementos() : Método para sumar las componentes de las parejas ordenadas
manejador_tupla : Instancia (objeto) de la clase ComprensionParejas

2.2 Herencia de clases y/o Polimorfismo

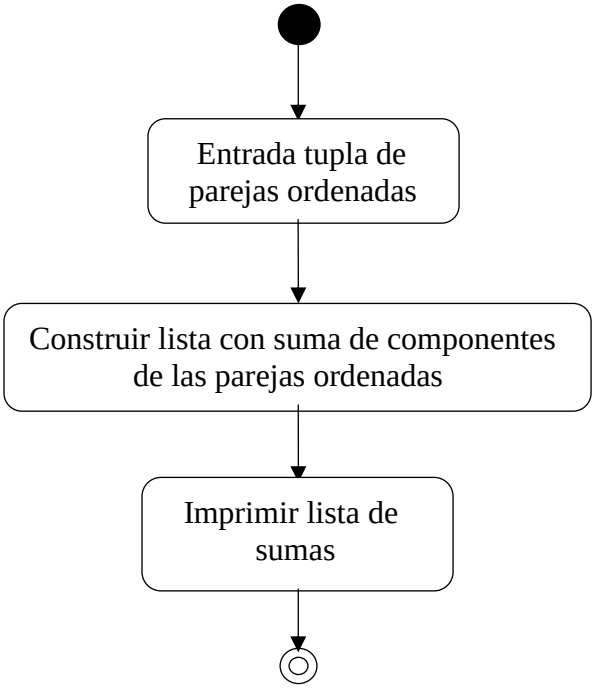
No se presenta herencia de clases.
No hay polimorfismo.

3. DESARROLLO DE LA SOLUCION PARA CADA OPERACION

3.1 Modelación Matemática
x + y (suma de parejas ordenadas)

3.2 Propuesta de solución

3.2.1 Diagrama de actividades




3.2.2 Definición de estructuras de datos
tupla_ejemplo: tupla

3.2.3 Desarrollo del algoritmo (con diagramas N -S)

| Inicio | Operaciones |
|---|---|
| Declaración de clases, variables y constantes<br><br>ComprensionParejas manejador_tupla<br>tupla | ComprensiónParejas:: sumar_elementos<br><br>retorne[x+y, para x,y en tupla]x % 2==0] |
| Algoritmo principal | |
| tupla_ejemplo= ((1,2),(3,4),(5,6),(7,8))<br>manejador_tupla=ComprensionParejas(tupla_ejemplo)<br>imprimir manejador_tupla.sumar_elementos | |

4.IMPLEMENTACION

```
# -*- coding: utf-8 -*-
"""
Creado Agosto 30 de 2023
nombre: manualPOOPython18
Que hace: Dada una tupla de parejas ordenadas, utilizar la
          comprensión de listas, para crear una lista con
          cada una de las sumas de las parejas ordenadas.
Autores: Hernando Alvarez R
         Leonardo Alvarez V
         Luis Fernando Alvarez V.
Universidad Tecnológica de Pereira

"""
class ComprensionParejas: # Clase principal
    def __init__(self, tupla): # self representa la
        self.tupla = tupla  # instancia de la clase
    # Sumar los dos elementos de las parejas ordenadas
    def sumar_elementos(self):
        return [x + y for x, y in self.tupla]

# Ejemplo de uso
def main():
    tupla_ejemplo = ((1, 2), (3, 4), (5, 6), (7, 8))
    manejador_tupla = ComprensionParejas(tupla_ejemplo)
    print("Suma de elementos:", manejador_tupla.sumar_elementos())
if __name__ == "__main__":
    main()
```

5.MANTENIMIENTO Y EVALUACION

Ejemplo No 19

Enunciado

Crear un ejemplo donde se aplique el concepto de herencia simple y se utilicen una clase base con una clase derivada.

1. IDENTIFICACION Y DELIMITACION DEL PROBLEMA A RESOLVER

1.1 Identificación

Introducir el concepto de herencia simple

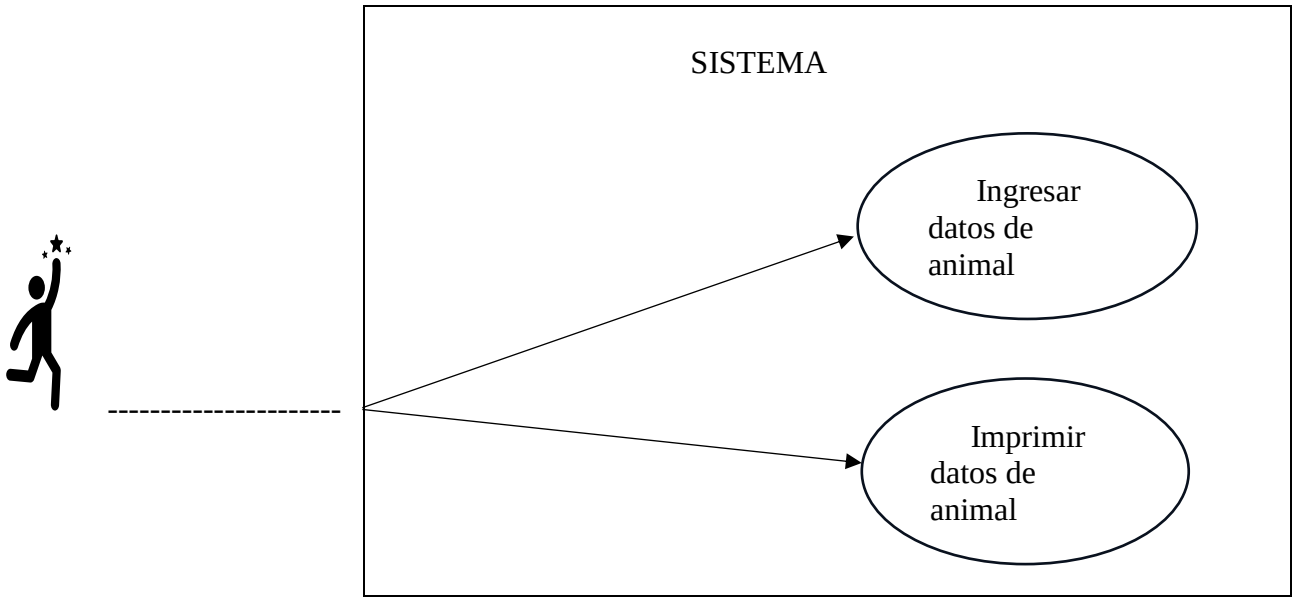
1.2 Delimitación

Utilizar solamente una clase base y una derivada

2. ANALISIS DEL PROBLEMA

(Representación de casos de uso e identificación de clases, usando UML)

CASOS DE USO



2.1 Identificación de las clases

| Animal # clase base | |
|---|---|
| Perro (Animal) # clase que hereda de animal<br>nombre<br>edad | :Animal<br>mi_perro: Perro |
| hacer_sonido() | |

2.1.1 Documentación de los miembros de la clase

nombre : Nombre del perro. Tipo cadena
edad : Edad del perro. Tipo cadena
hacer_sonido : Método para verificar la herencia
Animal : No tiene instancia de clase. Otra clase hereda de ella
Perro : Clase que hereda de la clase Animal
mi_perro : Instancia (objeto) de la clase Perro.

2.2 Herencia de clases y/o Polimorfismo

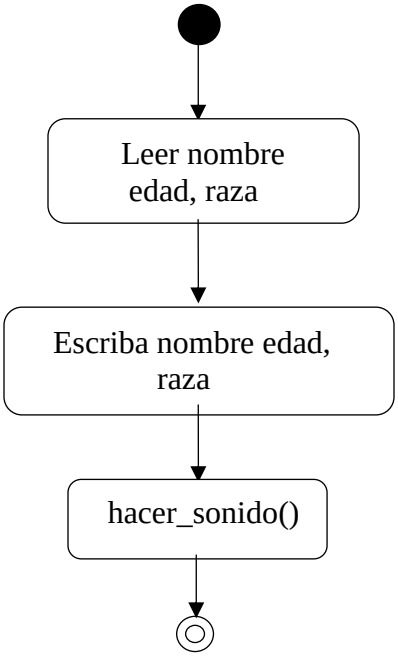
Perro hereda de la clase Animal

3. DESARROLLO DE LA SOLUCION PARA CADA OPERACION

3.1 Modelación Matemática
No se requieren cálculos

3.2 Propuesta de solución

3.2.1 Diagrama de actividades




3.2.2 Definición de estructuras de datos

3.2.3

No existen estructuras de datos.

3.2.3 Desarrollo del algoritmo (con diagramas N -S)

| Inicio | Operaciones |
|---|---|
| Declaración de clases, variables y constantes<br><br>Animal # clase base<br>nombre<br>edad<br><br>Perro (Animal) # Clasederivada<br>Super (nombre,edad)<br>raza | Animal :: hacer_sonido()<br>Pasar<br><br>Perro (Animal) :: hacer_sonido()<br>Retorne “Guauuuuuuuuu” |
| Algoritmo principal | |
| # crear una instancia de la clase Perro<br>mi_perro=Perro(nombre= “Perrolindo”,<br>edad=3, raza=“hermoso”)<br># Acceder a los atributos de la clase base<br>imprimir (nombre, raza, edad).<br># Llamar el método hacer_sonido de la clase derivada<br>Imprimir (mi_perro.hacer_sonido()) | |

4.IMPLEMENTACION

```
# -*- coding: utf-8 -*-
"""
Creado Agosto 30 de 2023
Nombre: manualPOOPython19
Que hace: Ejemplo para utilizar la herencia simple
Autores: Hernando Alvarez R
         Leonardo Alvarez V
         Luis Fernando Alvarez V.
Universidad tecnológica de Pereira
"""
# Definición de la clase base
class Animal:
    def __init__(self, nombre, edad): # self representa la
        self.nombre = nombre          # instancia de la clase
        self.edad = edad
    def hacer_sonido(self): # Método abstracto, será implementado
        pass                # en la clase derivada
# Clase derivada que hereda de Animal
class Perro(Animal):
   def __init__(self, nombre, edad, raza):
        # Llamamos al constructor de la clase
        # base usando super()
        super().__init__(nombre, edad)
        self.raza = raza
   def hacer_sonido(self):
         return "Guauuuuuuu ...."
```

```
# Ejemplo de uso
def main():
    # Crear una instancia de la clase Perro
    mi_perro = Perro(nombre="Perrolindo", edad=3, raza="Hermoso")
    # Acceder a los atributos de la clase base
    print(f"{mi_perro.nombre}--raza: {mi_perro.raza}--{mi_perro.edad}
años.")
    # Llamar al método hacer_sonido de la clase derivada
    print(mi_perro.hacer_sonido())
if __name__ == "__main__":
        main()
```

5.MANTENIMIENTO Y EVALUACION

Ejemplo No 20

Enunciado

Crear un ejemplo donde se aplique el concepto de polimorfismo y se utilice una clase base con tres clases derivadas.

1. IDENTIFICACION Y DELIMITACION DEL PROBLEMA A RESOLVER

1.1 Identificación

Con este ejemplo se pretende introducir el concepto de polimorfismo.

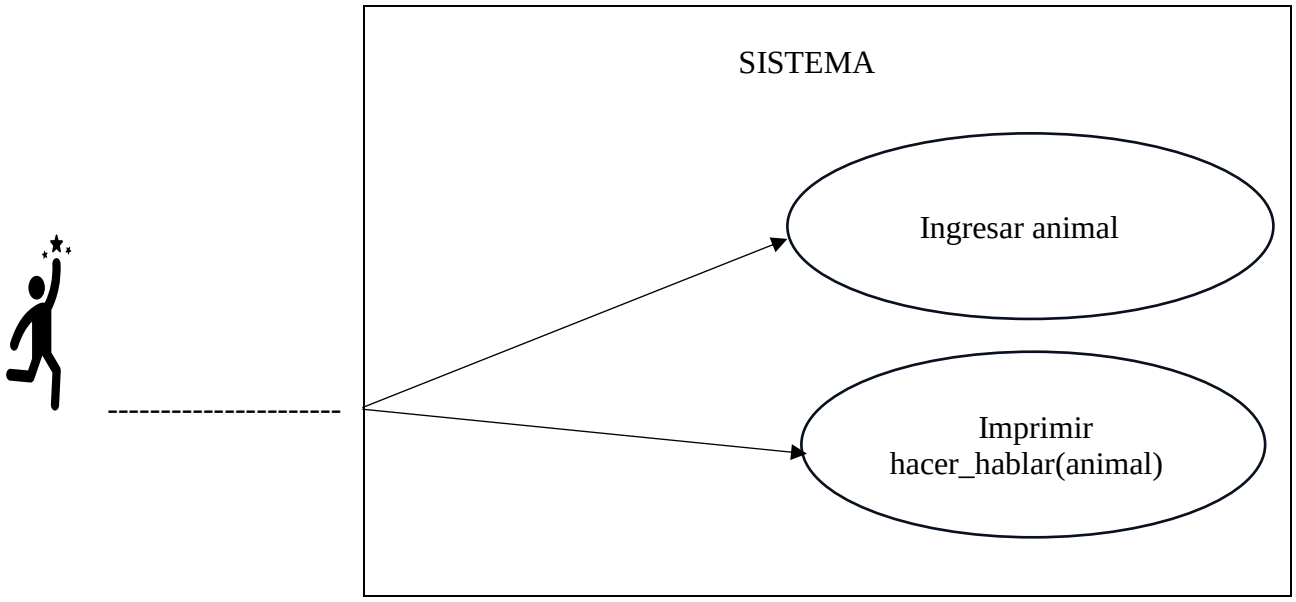
1.2 Delimitación

Se deben utilizar una clase base y tres clases derivadas

2. ANALISIS DEL PROBLEMA

(Representación de casos de uso e identificación de clases, usando UML)

CASOS DE USO



2.1 Identificación de las clases

| Animal #clase base | |
|---|---|
| Perro #clase derivada<br>Gato #clase derivada<br>Vaca #clase derivada | : Animal<br>perro: Perro<br>gato: Gato<br>vaca: Vaca |
| # La siguiente función no pertenece<br># a la clase base, ni a clases derivadas<br><br>hacer_hablar (animal) | |

2.1.1 Documentación de los miembros de la clase

hacer_hablar () : Método para generar el polimorfismo
: Animal (clase base).
perro : Instancia (objeto) de la clase derivada Perro.
gato : Instancia (objeto) de la clase derivada Gato.
vaca : Instancia (objeto) de la clase derivada Vaca.

2.2 Herencia de clases y/o Polimorfismo

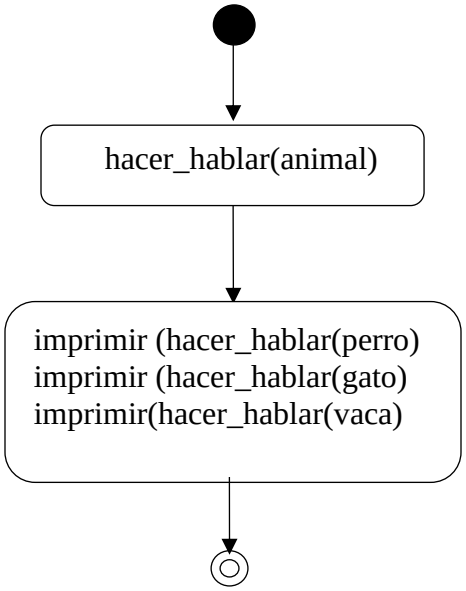
Existe polimorfismo con la clase base Animal y las tres clases derivadas: Perro, Gato y Vaca

3 DESARROLLO DE LA SOLUCION PARA CADA OPERACION

3.1 Modelación Matemática
No hay fórmulas matemáticas

3.2 Propuesta de solución

3.2.1 Diagrama de actividades




3.2.2 Definición de estructuras de datos

No existen estructuras de datos.

3.2.3 Desarrollo del algoritmo (con diagramas N -S)

| Inicio | Operaciones |
|---|---|
| Declaración de clases, variables y constantes<br><br>Animal # clase base<br>perro(Animal) Perro # clasederivada<br>gato(Animal) Gato # clasederivada<br>vaca(Animal) Vaca # clasederivada | Animal :: hablar ()<br>pasar<br>Perro(Animal) :: hablar()<br>retorne “Woof!:”<br>Gato (Animal) :: hablar()<br>retorne “Meow!”<br>Vaca (Animal) :: hablar()<br>retorne “Moo!”<br>Función que utiliza polimorfismo<br>hacer_hablar(animal)<br>retorne animal.hablar() |
| Algoritmo principal | |
| # crear instancia de cada clase derivada<br>perro= Perro()<br>gato= Gato()<br>vaca= Vaca()<br># utilizar la función haciendo<br># uso del polimorfismo<br>imprimir(hacer_hablar (perro)) # imprimira: ¡woof!<br>imprimir(hacer_hablar(gato)) # imprimira: ¡Meow!<br>imprimir(hacer_hablar(vaca)) # imprimira: ¡Moo! | |

## 4. IMPLEMENTACION

```
# -*- coding: utf-8 -*-
"""
Creado Agosto 30 de 2023
Nombre: manualPOOPython20
Que hace: Mostrar el manejo del polimorfismo
Autores: Hernando Alvarez R
         Leonardo Alvarez V
         Luis Fernando Alvarez V.
Universidad Tecnológica de Pereira
"""
# Definición de la clase base
class Animal:
    def hablar(self): # self representa la
        pass          # instancia de la clase
# Definición de una clase derivada
class Perro(Animal):
    def hablar(self):
        return"Woof!"
# Definición de una clase derivada
class Gato(Animal):
    def hablar(self):
        return"Meow"
# Definición de una clase derivada
class Vaca(Animal):
    def hablar(self):
        return"Moo"
# Función que utiliza el polimorfismo
def hacer_hablar(animal):
    return animal.hablar()
# Ejemplo de uso
def main():
    # Crear instancias de las clases derivadas
    perro = Perro()
    gato = Gato()
    vaca = Vaca()
    # Utilizar la función haciendo uso del polimorfismo
    print(hacer_hablar(perro))  # Imprimirá: Woof!
    print(hacer_hablar(gato))  # Imprimirá: Meow!
    print(hacer_hablar(vaca))  # Imprimirá: Moo!
if __name__ == "__main__":
    main()
# El polimorfismo es la capacidad de objetos
    # de diferentes clases para responder al mismo mensaje.
```

## 5.MANTENIMIENTO Y EVALUACION