Capítulo 1

Programación estructurada

1.1. Tipos de datos

El lenguaje Python ofrece una importante variedad de tipos de datos, de entre los cuales se destacan los siguientes:

Categoría	Tipo	Nombre	Uso
Texto	Cadena	str	Cadenas de caracteres de longitud arbitraria
Numérico	Entero	int	Números enteros
Numérico	Flotante	float	Números con punto flotante
Numérico	Complejo	complex	Números complejos
Lógico	Booleano	bool	Valores de verdad
Secuencia	Tupla	tuple	Secuencia inmutable de valores
Secuencia	Lista	lista	Secuencia mutable de valores
Secuencia	Rango	range	Secuencia inmutable de números enteros
Conjunto	Conjunto	set	Conjunto de valores sin repetición y con búsquedas rápidas
Asociación	Diccionario	dict	Pares ordenados clave / valor
Vacío	Sin tipo	none	Variables declaradas pero sin valor ni tipo

1.2. Operadores

Los datos de cada uno de los tipos del lenguaje pueden ser manipulados mediante un conjunto de operadores. Un operador es un símbolo o una palabra que realiza una operación sobre uno o más valores. Los valores pueden ser constantes, variables o expresiones. Los operadores pueden manipular una cantidad diferente de valores. La mayoría de los operadores se denominan binarios porque requieren dos operandos que es indican antes y después del operador. Asimismo existen algunos operadores unarios y un único operador ternario.

1.2.1. Operadores aritméticos

Requieren dos operandos numéricos de cualquier tipo. Si ambos operandos son del mismo tipo, es decir, dos enteros o dos flotantes el resultado generalmente será del mismo tipo de los operandos. En caso de que sean de tipos diferentes, el resultado será del tipo más grande. Por lo tanto, si se opera con un entero y un flotante, el resultado será flotante.

Símbolo	Uso	Observaciones
+	Suma	Aplicado a cadenas efectúa una concatenación
-	Diferencia	
*	Producto	Si un operando es una cadena y el otro es un entero, repite la cadena tantas veces como indique el operando numérico
/	Cociente	Siempre retorna un float
//	Cociente entero	Siempre retorna un int truncando los decimales
%	Módulo	Retorna el resto de una división
**	Potencia	Retorna el primer operando elevado a la potencia indicada por el segundo.

1.2.2. Operadores de asignación

Los operadores de asignación siempre son binarios, de los cuales el primer operando debe ser una variable o cualquier lugar donde se pueda guardar un valor, tal como un parámetro actual y el segundo debe ser una expresión. El tipo retornado siempre es el de la expresión asignada.

Símbolo	Uso	Observaciones
=	Asignación	
+=	Acumulación o incremento	El primer operando debe ser una variable a la cual se le incrementa su valor con el resultado de la expresión del segundo operando
-=	Decremento	
*= /=	Combinados	Todas las operaciones aritméticas pueden ser combinadas con asignación, aunque su uso es infrecuente

1.2.3. Operadores de comparación

Los operadores de comparación comparan los valores de dos operandos de un mismo tipo y retornan un valor de verdad. Si los tipos comparados son diferentes, se promueve uno de esos al tipo más amplio.

Símbolo	Uso	Observaciones
==	Igualdad	
!=	Diferencia	
>	Mayor que	
<	Menor que	
>=	Mayor o igual	
<=	Menor o igual	

Existe un operador de comparación cuya sintaxis y comportamiento son diferentes al resto. Es el único operador ternario del lenguaje y permite escribir como una expresión un cálculo que de otra forma requiere una instrucción condicional `if-else`. La sintaxis del operador es:

```
[valor si verdadero] if [condicion] else [valor si falso]
```

El operador evalua la condición, la cual debe evaluarse o convertirse a un valor de verdad, si la misma es verdadera se retorna el primer operando y en caso contrario el último.

```
importe = 45345
saldo = "Acreedor" if importe > 0 else "Deudor"
# La variable saldo finaliza con valor "Acreedor"

existe = False
print("Si" if existe else "No")
# Imprime "Si"
```

1.2.4. Operadores lógicos

Permiten efectuar operaciones de algebra booleana entre dos valores de verdad. Si los operandos no son booleanos los convierte evaluando los valores vacíos de cada tipo de datos como falsos, y cualquier otro valor como verdadero.

Símbolo	Uso	Observaciones
and	Y lógico	Aplica corto circuito, si el primer operando es falso, retorna falso sin evaluar el segundo
or	O lógico	Aplica corto circuito, si el primer operando es verdadero, retorna verdadero sin evaluar el segundo
not	Negación	Es un operador unario, invierte el valor de verdad indicado a continuación del mismo

1.3. Estructuras condicionales

1.3.1. Instrucción if

La instrucción if en Python permite ejecutar un bloque de código si se cumple una condición. La sintaxis general es:

```
if condicion:
    bloque de codigo
```

La condición puede ser una expresión lógica o booleana que se evalúa como verdadera (True) o falsa (False). Si la condición es verdadera, se ejecuta el bloque de código indentado después de los dos puntos (:). Si la condición es falsa, se salta el bloque de código y se continúa con el resto del programa.

Por ejemplo:

```
if x > 0:
    print("x es positivo")
```

En este caso, se imprime el mensaje ”x es positivo”solo si la variable x tiene un valor mayor que cero. De lo contrario, no se hace nada.

La instrucción if se puede combinar con else o elif para definir otros casos alternativos. Else se usa para ejecutar un bloque de código si la condición del if no se cumple. Elif se usa para evaluar otra condición después del if. Se pueden usar varios elif para crear múltiples ramas.

Por ejemplo:

```
if x > 0:
    print("x es positivo")
elif x < 0:
    print("x es negativo")
else:
    print("x es cero")
```

En este caso, se imprime un mensaje diferente según el valor de x. Si x es positivo, se imprime “x es positivo”. Si x es negativo, se imprime “x es negativo”. Si x es cero, se imprime “x es cero”.

1.3.2. Condiciones

Las instrucciones y cláusulas que evalúan condiciones exigen que la condición sea escrita como una expresión booleana. Como tal, una expresión puede ser una constante, una variable o el resultado de una operación.

Aunque es muy infrecuente, puede plantearse una condición con un valor True constante, pero esta variante tiene sentido únicamente si se plantea un ciclo infinito.

En cambio sí es muy habitual evaluar variables de tipo boolean, conocidas generalmente como banderas o centinelas. Para estas variables, la condición se redacta únicamente con el nombre de la variable. Durante la ejecución, el valor que la misma almacene servirá para ejecutar o no el bloque de la instrucción condicional. De esa manera si se dispone de una variable llamada `existe`, la redacción preferida de una instrucción if sera `if existe:`, siendo innecesario y hasta negativo el uso de una comparación con `if existe == True:`. De la misma manera, si se requiere evaluar que la variable contenga un valor falso, la redacción preferida es `if not existe:` en lugar de comprarar `if existe == False:`.

En las condiciones que involucren variables de tipos diferentes al boolean, la expresión a evaluar se redacta como una operación que utilice los operadores que devuelven valores de verdad, es decir, los operadores de comparación y los operadores lógicos.

1.3.3. Conversiones a boolean

Python ofrece un mecanismo de conversiones implícitas de todos los tipos de datos a boolean. Esta caracterísitica permite evaluar como verdaderas o falsas variables o expresiones de otros tipos aplicando ciertas reglas de conversión.

Esta característica se denomina *truthiness* y puede ser fuente de serios errores hasta que se la comprende acabadamente. La idea principal es que todo valor de un tipo diferente al boolean puede ser interpretado “como si fuera verdadero” (*truthy*) o “como si fuera falso” (*falsy*).

El criterio general es que se considera falsy todo dato “vacío”, dependiendo el valor exacto del tipo de datos. En el caso de los datos más simples el vacío es el más natural: 0 para los tipos numéricos, cadena vacía en el caso de las cadenas y None en el caso de las referencias a estructuras u objetos. Por otro lado, en los casos de las secuencias y estructuras de datos, se evalúan como falsy si están vacías, es decir, sin elementos.

Esta conversión implícita puede ser utilizada en ciertas condiciones para mejorar la legibilidad de la misma. De la misma manera en que para evaluar una bandera es preferible evitar la comparación con los valores True y False, se puede aprovechar la misma idea al evaluar si una lista está vacía o si una variable numérica es 0.

Con este criterio, las siguientes condiciones son válidas:

```
nombre = input("Ingrese su nombre")
edad = int(input("Ingrese su edad))

if not nombre: print("No ingresó su nombre")
if not edad: print("Debe ingresar una edad diferente a 0")

print("Su edad es " + ("impar" if edad % 2 else "par"))
```

1.3.4. Combinación de condiciones

Una característica particular del lenguaje python que siendo muy útil resulta difícil de aprender es la posibilidad que brinda para combinar operadores condicionales.

Una situación muy habitual es la de verificar si un valor numérico se encuentra en un rango. Para ello la solución más simple es la de comparar el valor con cada uno de los extremos del rango y conectar con el operador AND:

```
if nota > 0 and nota ≤ 10:
```

Pero dado que las condiciones que evalúan cada extremo del rango involucran a la misma variable y que ambas están conectadas por and, se pueden combinar de la siguiente forma:

```
if 0 < nota ≤ 10:
```

Debe prestarse especial atención a que esta posiblidad puede llevar a errores difíciles de detectar. Por ejemplo, es una situación razonable determinar que dos variables cumplan con la misma condición, por caso, que sean mayores a 5. La única forma adecuada es la de evaluar `if a > 5 and b > 5`. Pero es tentador intentar una combinación de la forma:

```
if a and b > 5:
```

La condición anterior, aunque es válida desde el punto de vista de la sintaxis del lenguaje, no evalúa lo que parece a simple vista: el operador and posee como primer operando una variable numérica la cual evalúa como truthy si es distinta de 0, por lo tanto dicha condición es equivalente a `if a ≠ 0 and b > 5`.

Otro inconveniente similar ocurre si se intenta combinar las condiciones como:

```
if a > b > 5:
```

En este otro caso ocurre algo similar, la sintaxis es correcta, pero el funcionamento es algo diferente. Esta condición efectivamente verifica que ambas variables sean mayores a 5, pero además exige que la segunda sea mayor a la primera, restricción que no estaba solicitada originalmente.

1.4. Estructuras repetitivas

1.4.1. Instrucción for

La instrucción for permite iterar sobre una secuencia de elementos, como un rango, una lista, una tupla o un diccionario. La sintaxis general de la instrucción for es la siguiente:

```
for elemento in secuencia:
    # hacer algo con el elemento
```

La variable elemento toma el valor de cada elemento de la secuencia en cada iteración del bucle. El bloque de código que se ejecuta en cada iteración debe estar indentado. El bucle termina cuando se recorre toda la secuencia o cuando se encuentra una instrucción break.

La instrucción for es utilizada para realizar tareas repetitivas con los elementos de una secuencia, como sumarlos, modificarlos o filtrarlos. También se puede usar para crear nuevas secuencias a partir de otras existentes, usando la comprensión de listas, tuplas o diccionarios.

Por ejemplo, para recorrer una lista se puede escribir el siguiente ciclo:

```
frutas = ["manzana", "banana", "naranja"]

# Por cada fruta de la lista de frutas:
for fruta in frutas:
    print(fruta)
```

1.4.2. Instrucción while

La instrucción “while” en Python se utiliza para crear un bucle o ciclo que se repite mientras se cumple una condición específica. A diferencia de la instrucción “for” que se utiliza para iterar sobre una secuencia conocida, la instrucción “while” se repite hasta que una condición se evalúa como falsa. La sintaxis general de la instrucción es la siguiente:

```
while condición:
   # bloque iterativo
```

En el ejemplo anterior “condición” es una expresión que se evalúa en cada iteración del bucle. Si la condición es verdadera, el bloque de código dentro del bucle se ejecuta. Si la condición es falsa el ciclo finaliza.

Es importante tener cuidado al utilizar la instrucción while para evitar que el bucle se convierta en un ciclo infinito. Para evitar esto, es común utilizar una lógica dentro del bucle que modifique las variables involucradas en la condición para que eventualmente se evalúe como falsa y el ciclo se detenga.

En el siguiente ejemplo se obtiene la suma de todos los dígitos de un número con operaciones aritméticas, por medio de la extracción del último dígito con el operador de módulo. En cada vuelta se desplazan los restantes digitos efectuando una división entera en 10. Dado que no necesariamente se conoce la cantidad de dígitos del número ingresado, se debe utilizar un ciclo while que dará una vuelta por cada dígito, y finaliza cuando el número se reduce a 0.

```
numero = int(input("Ingrese un número: "))
suma_digitos = 0

while numero > 0:
    # Obtiene el último dígito del número
    digito = numero % 10

    # Agrega el dígito a la suma total
    suma_digitos += digito

    # Elimina el último dígito del número
    numero //= 10

print("La suma de los dígitos es:", suma_digitos)
```

1.4.3. Saltos

La instrucción **break** se utiliza para terminar un bucle antes de que se complete su condición de finalización. Debe encontrarse dentro de alguna instrucción condicional que determine el momento en que debe interrumpirse el ciclo. La instrucción break solo afecta al bucle en el que se encuentra y no a los bucles anidados o externos.

La instrucción **continue** se utiliza dentro de ciclos para omitir el resto del código en una iteración actual y pasar

a la siguiente iteración. Cuando se encuentra una instrucción continue, el flujo de ejecución del programa salta inmediatamente al principio del ciclo, sin ejecutar el código restante de esa iteración en particular.

1.4.4. Cláusula else

La cláusula `else` en los ciclos se utiliza para especificar un bloque de código que se ejecuta cuando el ciclo ha terminado de iterar sobre todos los elementos de una secuencia o cuando la condición del ciclo se evalúa como falsa. La cláusula else se ejecuta después de que el ciclo ha finalizado de forma normal, es decir, sin interrupciones como break o return.

En el siguiente ejemplo se verifica la existencia de la letra A en una cadena. En el momento en que se encuentra la primera aparición se interrumpe el ciclo con break para no continuar con el recorrido. Por lo tanto, que el ciclo finalice normalmente, sin la interrupción, significa que la letra no fue encontrada:

```
texto = input("Ingrese un texto: ")

for letra in texto:
    if letra == "A":
        print("Contiene la letra A")
        break
else:
    print("No contiene la letra A")
```

1.5. Funciones

1.5.1. Sintaxis

Las funciones se definen utilizando la palabra clave `def` seguida del nombre de la función. Luego del nombre deben existir un par de paréntesis que pueden contener los parámetros de la función y el símbolo de dos puntos (:). A continuación, se escribe el bloque de código de la función indentado.

```
def nombre_de_funcion(parametro1, parametro2, ...):
    # Bloque de código de la función
    # Puede incluir declaraciones, operaciones y retornos
```

En el siguiente ejemplo se define una función con retornos y parámetros:

```
def calcular_area_triangulo(base, altura):
   area = (base * altura) / 2
   return area
```

La función llamada `calcular_area_triangulo` que toma dos parámetros base y altura. Dentro del bloque de código de la función, se calcula el área de un triángulo utilizando la fórmula correspondiente y se almacena en la variable area. Luego, se utiliza la palabra clave `return` para devolver el valor calculado de area como resultado de la función.

Después de definir una función, puede ser invocada desde otro lugar del código utilizando su nombre y proporcionando los argumentos necesarios. De esta manera la función anterior puede ser invocada como:

```
resultado = calcular_area_triangulo(5, 3)
print("El área del triángulo es:", resultado)
```

En el caso de que una función no incluya una instrucción return o posea una instrucción return sin indicar ningún valor de retorno, la misma retorna automáticamente un valor de None al finalizar su ejecución.

1.5.2. Parámetros

Existen diversas formas de indicar los parámetros de una función, cada una de las cuales tiene una utilidad bien marcada.

Parámetros posicionales: Son los parámetros que se pasan a la función en el mismo orden en el que se definen en la firma de la función. Estos parámetros son obligatorios y deben ser proporcionados al llamar a la función.

```
def saludar(nombre, edad):
    print("Hola", nombre, "tenés", edad, "años.")

saludar("Olga", 25)
# Ejemplo de llamada con parámetros posicionales
```

Parámetros con valor predeterminado: Son parámetros que tienen un valor asignado por defecto en caso de que no se les pase un valor al llamar a la función. Estos parámetros son opcionales.

```
def saludar(nombre, edad=30):
    print("Hola", nombre, "tenés", edad, "años.")

saludar("Jorge")
# Ejemplo de llamada sin proporcionar el parámetro "edad"
```

Parámetros de palabra clave: Se especifican durante la llamada a la función utilizando el formato `nombre_parametro=valor`. Estos parámetros son opcionales, por lo tanto permiten omitir aquellos que posean un valor por defecto. Asimismo permiten especificar los argumentos en cualquier orden.

```
def saludar(nombre, edad):
    print("Hola", nombre, "tenés", edad, "años.")

saludar(edad=40, nombre="Carlos")
# Ejemplo de llamada con parámetros de palabra clave
```

Parámetros variables (*args): Permite pasar un número variable de argumentos posicionales a una función. Los argumentos se agrupan en una tupla dentro de la función.

```
def sumar(*numeros):
   total = 0
   for num in numeros:
      total += num
   return total

resultado = sumar(1, 2, 3, 4, 5)
# Ejemplo de llamada con parámetros variables
```

Parámetros de palabras clave variables (kwargs):** Permite pasar un número variable de argumentos de palabra clave a una función. Los argumentos se agrupan en un diccionario dentro de la función.

```
def imprimir_datos(**datos):
   for clave, valor in datos.items():
      print(clave + ":", valor)

imprimir_datos(nombre="Juana", edad=25, ciudad="Pinamar")
# Ejemplo de llamada con parámetros de palabras clave variables
```

1.5.3. Retorno

Las funciones pueden retornar cero, uno o más expresiones como resultados. En el caso de no incluir ninguna instrucción return, u omitir el valor retornado, la misma entrega un valor de None.

Para que una función retorne más de un valor, se los debe indicar separados por comas. En la invocación a la misma los valores retornados pueden ser asignados a una serie de variables también separadas por comas.

Por ejemplo, la siguiente función recibe como parámetro una lista de números y devuelve la sumatoria de los mismos y su promedio:

```
def promedio_suma(numeros):
    suma = 0

    for x in numeros:
        suma += n
    promedio = suma / len(numeros)

    return promedio, suma

lista = [10, 20, 30, 40, 50]
promedio, suma = promedio_suma(lista)

print("Promedio:", promedio)
print("Suma:", suma)
```

1.6. Ejercicios

1.6.1. Estación de servicio

Una estación de servicio que dispone de 10 surtidores y necesita gestionar información relacionada con la venta de combustibles en la jornada.

De cada surtidor se conoce:

- Número de Surtidor (validar que sea un número entre 1 y 30)
- Cantidad: representa la cantidad de litros de combustible vendido por el surtidor (validar que sea un número positivo)
- Tipo: representa el tipo de combustible del surtidor. Los valores que puede asumir son 1 representa “Nafta Super”, 2 representa “Nafta Especial” y 3 representa “Gasoil” (validar que se ingresen valores válidos).

Se pide calcular e imprimir:

- El total de litros vendidos en la jornada, por tipo de combustible.
- El número de surtidor que menos combustible vendió.
- El promedio por surtidor en litros de combustible vendido en la jornada (promedio general, es un único resultado).

estacion.py

```
total_nafta_super = total_nafta_especial = total_gasoil = 0
menor = None
surtidor_menor = None
total = 0

for i in range(10):
    numero = int(input('Ingrese número de surtidor (1 a 30): '))
    while not 0 < numero ≤ 30:
        print('Ingresó un número inválido')
        numero = int(input('Ingrese número de surtidor (1 a 30): '))

    cantidad = int(input('Ingrese la cantidad de litros vendidos: '))
    while cantidad < 0:
        print('Debe ingresar un número positivo')
        cantidad = int(input('Ingrese la cantidad de litros vendidos: '))

    tipo = int(input('Ingrese el tipo de combustible (1 a 3): '))
    while not 1 ≤ tipo ≤ 3:
        print('Ingresó un número inválido')
        tipo = int(input('Ingrese el tipo de combustible (1 a 3): '))

    if tipo == 1:
        total_nafta_super += cantidad
    elif tipo == 2:
        total_nafta_especial += cantidad
    else:
        total_gasoil += cantidad

    if not menor or cantidad < menor:
        menor = cantidad
        surtidor_menor = numero

total = total_nafta_super + total_nafta_especial + total_gasoil
promedio = total // 10

print(f"Total de litros de nafta super: {total_nafta_super}")
print(f"Total de litros de nafta especial: {total_nafta_especial}")
print(f"Total de litros de gasoil: {total_gasoil}")
print(f"Surtidor que menos litros vendió: {surtidor_menor}")
print(f"Promedio de litros por surtidor: {promedio}")
```

1.6.2. Procesamiento de temperaturas en una lista

Ingresar un conjunto de temperaturas en una lista, finalizar la carga cuando se reciba un 50. Sólo aceptar temperaturas entre -20 y 49 grados.

Calcular y mostrar:

- Cantidad de días con temperatura bajo cero
- Promedio de temperaturas
- Promedio de temperaturas de los días cálidos, es decir con temp. mayor a 20
- Mostrar “Si” o “No” para indicar si hubo algún día con más de 40 grados.
- La mayor temperatura de los días que no fueron cálidos
- Cantidad de días con temperatura menor al promedio

validacion.py

```python
def ingresar_numero_entre(mensaje, minimo, maximo):

    valor = float(input(mensaje))
    while not minimo ≤ valor ≤ maximo:
        print(f"Debe ingresar un valor entre {minimo} y {maximo}")
        valor = float(input(mensaje))

    return valor
```

procesolistas.py

```python
def cantidad_menor(lista, techo):
    """
    Cuenta los elementos de una lista que sean menores a un tope

    :param lista: lista de valores
    :type lista: lista de datos que soporte comparación por menor
    :param techo: valor máximo para filtrar
    :type techo: el mismo de los elementos de la lista
    :returns: la cantidad de elementos cuyo valor sea menor al techo
    :rtype: entero
    """
    c = 0
    for x in lista:
        if x < techo:
            c += 1

    return c


def promedio(lista):
    """
    Calcula el promedio simple de todos los elemento de una lista

    :param lista: lista
    :type lista: lista de números
    :returns: el promedio de todos los valores de la lista o 0 si la lista está vacía
    :rtype: flotante
    """
    cantidad = len(lista)
    if cantidad == 0:
        return 0

    suma = 0
    for x in lista:
        suma += x

    return suma / cantidad


def existe(lista, buscado):
    """
    Busca un valor en una lista e informa si lo pudo encontrar

    :param lista: una lista de cualquier tipo
    :param buscado: el valor que se busca
    :tipo buscado: el mismo tipo que el de los datos almacenados en la lista

    :returns: verdadero si el elemento buscado existe en la lista y falso en caso contrario
    :rtype: boolean
    """
    for x in lista:
        if x == buscado:
            return True

    return False
```

temperaturas.py

```
from validacion import *
from proceso_listas import *

def cargar_temperaturas():

    temperaturas = []

    print("Ingrese las temperaturas entre -20 y 49. Finaliza con 50.")
    t = ingresar_numero_entre("Ingrese una temperatura: ", -20, 50)
    while t ≠ 50:
        temperaturas.append(t)
        t = ingresar_numero_entre("Ingrese una temperatura: ", -20, 50)

    return temperaturas


def promedio_mayores(temperaturas, piso):

    cantidad = 0
    suma = 0

    for x in temperaturas:
        if x > piso:
            suma += x
            cantidad += 1

    if cantidad == 0:
        promedio = 0
    else:
        promedio = suma / cantidad
    #promedio = suma / cantidad if cantidad ≠ 0 else 0

    return promedio


def existe_mayor(temperaturas, piso):

    for x in temperaturas:
        if x > piso:
            return True

    return False



def mayor_dias_calidos(temperaturas):

    mayor = None

    for x in temperaturas:
        if x ≤ 20:
            if mayor is None or x > mayor:
                mayor = x

    return mayor


def calcular(temperaturas):

    dias_bajo_cero = cantidad_menor(temperaturas, 0)
    promedio_todos = promedio(temperaturas)
    promedio_dias_calidos = promedio_mayores(temperaturas, 20)
    hubo_40_grados = existe_mayor(temperaturas, 40)
    mayor = mayor_dias_calidos(temperaturas)
    dias_menor_al_promedio = cantidad_menor(temperaturas, promedio_todos)


    print(f"Hubo {dias_bajo_cero} días con temperatura bajo cero")
    print(f"El promedio de temperaturas fue de {promedio_todos}")
    if (promedio_dias_calidos ≠ 0):
        print(f"Y de los días cálidos fue de {promedio_dias_calidos}")

    print("¿Hubo días con más de 40 grados?", "Si" if hubo_40_grados else "No")
    if mayor:
        print(f"La mayor temperatura de los días que no fueron cálidos fue de {mayor}")
```

```
    else:
        print("Todos los días fueron cálidos")

    print(f"Hubo {dias_menor_al_promedio} días con temperatura menor al promedio")


def principal():

    temperaturas = cargar_temperaturas()
    calcular(temperaturas)


if __name__ == "__main__":
    principal()
```

Capítulo 2

Secuencias

2.1. Introducción

Los tipos de datos de secuencias ofrecen la posibilidad de almacenar un conjunto de datos con un único identificador, para poder acceder a los mismos posteriormente en tanto forma individual como grupal. Asimismo, la instrucción for permite recorrer cualquier tipo de secuencia, entregando en cada iteración cada uno de los valores almacenados en la misma.

2.1.1. Tamaño

Se puede obtener el tamaño de una secuencia con la función `len`, pasando la misma como parámetro. La función retorna un valor entero indicando la cantidad de elementos almacenados en la secuencia.

2.1.2. Operador de acceso indexado

Las secuencias disponen del operador de acceso indexado, que se indica como un par de corchetes a continuación del identificador. El mismo permite obtener el valor almacenado en una posición específica de la secuencia, identificando cada una de ellas mediante un número natural que inicia en 0. De esta manera, el primer elemento posee el índice 0, el segundo el 1, etc.

También pueden utilizarse índices negativos para acceder a los elementos desde el último, indicando como -1 al último, -2 al penúltimo y así sucesivamente.

El índice suele redactarse con una constante o variable entera, pero también pueden usarse expresiones.

```
titulo = "Paisaje"

print(titulo[2]) # imprime "i"
print(titulo[-2]) # imprime "j"
print(titulo[len(titulo)//2]) # imprime "s"
```

2.1.3. Operador de rebanadas

El operador de rebanadas o *slicing* también se representa mediante corchetes ([]), pero incluye un rango de índices separados por dos puntos (:). Permite extraer un subconjunto de elementos contiguos de una secuencia. Su retorno siempre será una nueva secuencia que copia los elementos de la original sin modificarla.

El mismo permite indicar un índice inicial y otro final del la sección a rebanar. Si no se indica el inicial, se asume 0; mientras que si no se indica el final, se asume el último. El corte de la secuencia siempre incluye al valor del índice inicial pero no incluye al índice final.

Por ejemplo:

```
titulo = "Paisaje"

print(titulo[1:3]) # imprime "ai"
print(titulo[:3]) # imprime "Pai"
print(titulo[3:]) # imprime "saje"
```

Una variante del operador de rebanadas permite indicar un salto al seleccionar los elementos a cortar. Para ello se puede incluir un segundo símbolo de dos puntos y a continuación un número entero estableciendo que se corten los elementos desde el índice inicial y se salteen los siguientes. Así, con un salto de 2, se extraen las posiciones alternadas, mientras que con un salto de -1 se recorre la secuencia hacia atrás. De esta manera, utilizando una rebanada sin indicar inicio ni final y con salto -1, se obtiene una nueva secuencia inviertiendo la primera.

```
titulo = "Paisaje"

print(titulo[::2]) # imprime "Piae"
print(titulo[5:0:-1]) # imprime "jasia"
print(titulo[::-1]) # imprime "ejasiaP"
print(titulo[::-2]) # imprime "eaiP"
```

2.2. Cadenas

La secuencia más simple es la secuencia de caracteres, tambien denominada cadena de caracteres o simplemente cadena. Las mismas almacenan un conjunto de caracteres como un texto, los cuales pueden ser manipulados en su conjunto o en forma individual.

Para indicar un valor constante se las delimita con comillas simples (') o dobles ("), pero utilizando el mismo símbolo tanto en la apertura como en el cierre.

Las variables de tipo cadena pueden ser asignadas y consultadas como variables para operar con el texto como una unidad, pero tambien puede accederse a cada caracter que conforma la secuencia mediante los operadores de acceso indexado y de rebanadas. El operador de suma (+) une dos cadenas y el operador de multiplicacion repite una cadena una cantidad de veces indicada en el segundo operando.

A continuación se presentan algunas de las operaciones más habituales:

```
mensaje1 = "Hola"
mensaje2 = "mundo!"

# Concatenación
mensaje_concatenado = mensaje1 + " " + mensaje2
print(mensaje_concatenado)  # Resultado: 'Hola mundo!'

# Multiplicacion
print("A" * 10) # Resultado: AAAAAAAAAA

# Obtener una subcadena utilizando rebanada
subcadena = mensaje1[1:3]
print(subcadena)  # Resultado: 'ol'

# Convertir a mayúsculas
mayusculas = mensaje1.upper()
print(mayusculas)  # Resultado: 'HOLA'

# Convertir a minúsculas
minusculas = mensaje2.lower()
print(minusculas)  # Resultado: 'mundo!'
```

2.2.1. Otras operaciones

Nombre	Uso
startswith	Indica si la cadena comienza con una subcadena pasada por parámetro
endswith	Indica si la cadena termina en una subcadena pasada por parámetro
find	Busca una subcadena e informa la posición en que se encuentra
join	Concatena todos los elementos de una secuencia con un delimitador
replace	Reemplaza todas las apariciones de una subcadena por otra
strip	Elimina todos los espacios que se encuentren al inicio y al final
split	Divide la cadena con un delimitador y devuelve una lista con los valores extraidos

2.2.2. Cadenas con formato

Cuando se requiere concatenar valores constantes con valores de variables, la operación de concatenación es impráctica porque no permite especificar fácilmente el formato de cada variable. Frecuentemente se requiere incluir dentro de una cadena larga el valor de una o más variables pero indicando características para la presentación de tales variables como ancho o alineación.

Para ello se dispone de las cadenas formateadas o *f-strings*, las cuales permiten agregar dentro de la cadena de *placeholders* o zonas de reemplazo, indicadas con un par de llaves {}, dentro de cada uno de los cuales se ubica el valor de una variable u operación. Para indicar que una cadena incluye placeholders debe indicarse con una letra f antes de la comilla de apertura.

Por ejemplo, para incorporar el valor de las variables `nombre` y `apellido` dentro de una cadena que contenga un saludo, en lugar de realizar una concatenación se las puede ubicar dentro de la cadena en un placeholder para cada una:

```
saludo = "Hola " + apellido + ", " + nombre + "! ¿cómo estás?"
saludo = f"Hola {apellido}, {nombre}! ¿cómo estás?"
```

A cada valor incrustado se le puede indicar un tamaño mínimo, de forma tal que si el valor no lo cumple se complete con espacios hasta alcanzar el tamaño indicado. Asimismo, se puede especificar una dirección de alineación para que los espacios sean agregados en uno o ambos extremos del valor y que el mismo quede alineado a la izquierda, a la derecha o centrado.

De esta manera, en cada placeholder se puede especificar:

```
f"{expresion:direccion ancho}"
```

La dirección se especifica con los siguientes símbolos

Símbolo	Dirección
>	Alineado a la derecha, agrega espacios antes del valor
<	Alineado a la izquierda, agrega espacios después del valor
^	Centrado, agrega espacios antes y después del valor

En el caso de las expresiones de tipo cadena de caracteres, la alineación por defecto es a la izquierda, mientras que para los valores numéricos la alineación por defecto es a la derecha.

Por ejemplo:

```
print(f"Hola {apellido:>20}, {nombre}, como estás?")
'Hola               Perez, Juan, como estás?'

print(f"Hola {apellido:<20}, {nombre}, como estás?")
'Hola Perez               , Juan, como estás?'

print(f"Hola {apellido:^20}, {nombre}, como estás?")
'Hola        Perez        , Juan, como estás?'
```

La especificación del ancho requiere un número entero, excepto en el caso de la presentación de valores de tipo float. Para esta situación el ancho se indica con un número entero indicando el ancho total, incluyendo la parte entera, el punto de decimal y la parte decimal, seguidos por un punto y la cantidad de dígitos que deben presentarse a la derecha del punto decimal y una letra f. De esta manera, para mostrar una variable con dos decimales se debe especificar {`variable:8.2f`}, y la misma se presenta con 8 caracteres en total, los cuales se reparten con cinco para la parte entera, el punto decimal y dos para los decimales.

2.3. Tuplas

La tupla es un tipo de dato que se utiliza para almacenar una secuencia ordenada e inmutable de elementos. A diferencia de las listas, que se definen utilizando corchetes ([]), las tuplas se definen utilizando paréntesis (()) o simplemente separando los elementos por comas. Por ejemplo:

```
tupla1 = (1, 2, 3)
tupla2 = "a", "b", "c"
```

En estos ejemplos, tanto tupla1 como tupla2 son variables que contienen tuplas.

Las tuplas son similares a las listas en el sentido de que pueden contener múltiples elementos, pero tienen la particularidad de ser inmutables, lo que significa que no se pueden modificar una vez creadas. Esto implica que no se pueden agregar, eliminar o cambiar elementos individualmente en una tupla.

Se pueden acceder a los elementos individuales de una tupla utilizando el operador de acceso indexado. Por ejemplo:

```
tupla3 = (10, 20, 30)
primer_elemento = tupla3[0]  # Acceso al primer elemento de la tupla
print(primer_elemento)  # Resultado: 10
```

En este ejemplo, se utiliza el operador de acceso indexado para acceder al primer elemento de la tupla “tupla3” y se almacena en la variable “primer_elemento”.

Las tuplas son útiles cuando se requiere almacenar una colección de elementos que no deben cambiar, como coordenadas de un punto, información fija o estructuras de datos que no deben modificarse accidentalmente.

Cuando una función retorna una serie de valores separados por comas, en realidad está devolviendo una tupla. Desde la llamada a la misma se puede asignar dicho retorno en una variable que tomará el tipo de datos tupla, o en una serie de variables separadas por comas, de forma tal que cada una de ellas será asignada con cada uno de los valores integrantes de la tupla. Esto se logra mediante una característica del lenguaje denominada desestructuración.

2.3.1. Desestructuración

La desestructuración es una herramienta que ofrece el lenguaje para poder asignar más de una variable a la izquierda del operador de asignación, de forma tal que cada una de tales variables sea asignada con cada uno de los valores de una secuencia que se presente a la derecha del operador.

De esta manera, son válidas las siguientes asignaciones:

```
x, y = 23, 44
# x = 23
# y = 44

precios = [58,22,99]
pre1, pre2, pre3 = precios
# pre1 = 58
# pre2 = 22
# pre3 = 99

letra1, letra2, letra3 = "DAO"
# letra1 = D
# letra2 = A
# letra3 = O
```

Este mecanismo permite que si una función retorna más de un valor los mismos puedan ser asignados en variables individuales al retornar la misma. También permite realizar operaciones que de otra forma requeririan varias operaciones de asignación o incluso variables auxiliares. Es notable el caso del intercambio de dos variables, que puede resolverse de una manera muy simple mediante `a, b = b, a`.

La cantidad de variables asignadas mediante desestructuración debe coincidir con el tamaño de la secuencia. En el caso de que la secuencia asignada tenga una cantidad grande o variable de elementos la desestructuración puede realizarse dejando que alguna de las variables sea indicada con un `*`. De esta manera, en las otras variables se asignan valores individualmente mientras que en la indicada con el asterisco se guarda una nueva secuencia con los valores restantes:

```
persona = "Juan", "Perez", 34, "San Martin 2423", "5000"
nombre, apellido, *otros_datos = persona
# nombre = "Juan"
```

```
# apellido = "Perez"
# otros\_datos =  34, "San Martin 2423", "5000"
```

2.4. Listas

Una lista es un tipo de dato que se utiliza para almacenar una colección ordenada y mutable de elementos. Se definen utilizando corchetes ([]), y los elementos de la lista se separan por comas. Por ejemplo:

```
lista1 = [1, 2, 3]
lista2 = ['a', 'b', 'c']
lista3 = [0] * 20
```

En estos ejemplos, tanto `lista1` como `lista2` son variables que contienen listas. Se deben utilizar corchetes y separar los elementos por comas para definir una lista. Incluso se puede declarar una lista vacía con los corchetes y sin indicar ningún valor entre ellos. La listad denominada `lista3` esta generada con el operador de producto, el mismo genera una lista de 20 elementos, todos con valor 0.

Las listas son similares a las tuplas, pero tienen la ventaja de ser mutables, lo que significa que se pueden modificar una vez creadas. Esto implica que se puede agregar, eliminar o cambiar elementos individualmente en una lista. Asimismo se puedes acceder a los elementos individuales de una lista utilizando el operador de acceso indexado.

Además de acceder a elementos individuales, las listas en Python admiten una variedad de operaciones y métodos que te permiten modificar, agregar, eliminar y manipular los elementos de la lista. Algunas operaciones comunes incluyen:

```
lista = [1, 2, 3]

#Agregar elementos a una lista con el método append():
lista.append(4)
print(lista)  # Resultado: [1, 2, 3, 4]

#Eliminar elementos de una lista con el método remove():
lista.remove(2)
print(lista)  # Resultado: [1, 2, 4]

# Obtener la longitud de una lista con la función len():
longitud = len(lista)
print(longitud)  # Resultado: 3

# Revertir una lista con el método reverse():
lista.reverse()
print(lista)  # Resultado: [4, 2, 1]

# Ordenar una lista con el método sort():
lista.sort()
print(lista)  # Resultado: [1, 2, 4]
```

2.5. Generación de listas por comprensión

La generación por comprensión, también conocida como comprensión de listas (list comprehension en inglés), es una construcción sintáctica que permite crear listas de manera concisa y eficiente basándose en una expresión y un conjunto de iteraciones o condiciones.

La sintaxis básica de una comprensión de lista es la siguiente:

```
nueva_lista = [expresión for elemento in secuencia]
```

Donde `expresión` representa la expresión o cálculo que se realizará en cada elemento de la secuencia, `elemento` es la variable de iteración que toma el valor de cada elemento en la secuencia, y `secuencia` es la fuente de valores sobre la cual se iterará.

Por ejemplo, dada una lista de números la necesidad de crear una nueva lista que contenga el cuadrado de cada número. Se puede usar una comprensión de lista de la siguiente manera:

```
numeros = [1, 2, 3, 4, 5]
cuadrados = [numero ** 2 for numero in numeros]
print(cuadrados)
# [1, 4, 9, 16, 25]
```

En este ejemplo, la comprensión de lista `[numero ** 2 for numero in numeros]` itera sobre cada elemento `numero` en la lista `numeros` y calcula el cuadrado de cada número utilizando la expresión `numero ** 2`. Los resultados se agregan automáticamente a una nueva lista llamada `cuadrados`.

Además de las iteraciones simples, también es posible incluir condiciones en una comprensión de lista. Por ejemplo, si se desea obtener solo los números pares de una lista, se puede agregar una condición utilizando la cláusula if:

```
numeros = [1, 2, 3, 4, 5]
pares = [numero for numero in numeros if numero % 2 == 0]
print(pares)
# [2, 4]
```

En este caso, la comprensión de lista filtra los números de la lista pero solo incluye aquellos que cumplen la condición, es decir, los números pares.

2.6. Ejercicios

2.6.1. Simulador de ruleta

Se necesita desarrollar un programa que simule el juego de la ruleta.
Para ello se deben generar al azar 1000 tiradas y luego informar:

- Cantidad de pares e impares
- Cantidad de tiradas por cada docena
- Porcentaje de ceros sobre el total de jugadas.
- Cantidad de rojos y de negros

Solución La simulación está programada en la función `simulacion`. Dado que los resultados son numerosos, la función no los retorna sino que los imprime directamente al terminar los cálculos.

Para el conteo de tiradas clasificado por color, el problema principal es el de conocer el color de cada número en una ruleta real. El algoritmo de determinación del color es el siguiente:

- Los números 10 y 28 son negros.
- Los otros números son negros si la suma de sus dígitos es par.
- La suma de los digitos no es la suma simple, si al sumar los dígitos del número el resultado es mayor a 9, deben volver a sumarse los digitos de dicho resultado hasta reducir a un único dígito.

Para ello el programa inicia generando una lista con los colores de los números, almacenando en la misma una tupla en la que el primer elemento es un número y el segundo el color asociado al mismo. En dicha lista las tuplas se almacenan de forma que cada número está almacenado en la lista en su mismo índice, por ejemplo, el número 10 esta almacenado como (10,"N") en la posición 10. La lista es creada en la función `generar_ruleta`.

Para el cálculo del color la función `get_color` recibe un número y devuelve el color correspondiente al parámetro.

Finalmente la función `reducir_numero` recibe un número entero y calcula la suma de sus dígitos. La función se llama recursivamente cuando la suma obtenida es mayor a 9 y por lo tanto es un número de más de un dígito.

ruleta.py

```
"""
Simulador de Ruleta

Desarrollar un programa que simule el juego de la ruleta.

Para ello generar al azar 1000 tiradas y luego informar:

* Cantidad de pares e impares
* Cantidad de tiradas por cada docena
* Porcentaje de ceros sobre el total de jugadas.
* Cantidad de rojos y de negros
"""

import random
from rich import print
from rich.console import Console
from rich.panel import Panel
from rich.table import Table

"""
Reglas:

Comienza en 1,R (1-ROJO)
Los números negros N son aquellos cuya reducción es PAR, o sea, la suma de sus dígitos es divisible por 2
"""

console = Console()

def reducir_numero(numero: int) -> int:
    """La función toma un número entero y devuelve un entero que es igual a la suma de todos sus dígitos.
    Si la suma tiene dos dígitos repite el proceso hasta que tenga un solo dígito.

    :param numero: Número que se desea reducir
    :type numero: int
    :return: Valor de UN dígito, obtenido a partir de la suma de los dígitos del número original.
    :rtype: int
```

```
    """

    # Caso base: Si el número tiene un solo dígito, devolverlo tal cual
    if numero < 10:
        return numero
    # Convertir el número en una lista de dígitos
    digitos = [int(digito) for digito in str(numero)]
    # Calcular la suma de los dígitos
    suma_digitos = sum(digitos)
    # Llamar recursivamente a la función con la suma de los dígitos
    return reducir_numero(suma_digitos)

def get_color (nro: int) -> str:
    """Dado un número de la ruleta, determinar el color de dicho número en el paño

    :param nro: Número del cual se desea averiguar el color
    :type nro: int
    :return: Color del número indicado. Valores válidos: V (verde), R (rojo), N (negro)
    :rtype: str
    """
    if nro == 0:
        return 'V'
    elif nro == 10 or nro == 28:
        return 'N'
    else:
        # Los numeros negros son los pares
        # Aquellos cuya reducción es par
        # El 10 y el 28
        if reducir_numero(nro)%2 == 0:
            return 'N'
        else:
            return 'R'

def generar_ruleta():
    """Genera un tablero de ruleta, como una lista de tuplas conteniendo (numero,color)
    """
    ruleta = []
    for nro in range (0,37):
        ruleta.append((nro,get_color(nro)))
    return ruleta

def imprimir_ruleta(rul: list[(int,str)]):
    """Imprime por consola un paño de ruleta, obtenido por medio de una lista de tuplas

    :param rul: lista que representa el paño de la ruleta
    :type rul: lista de tuplas (numero, color)
    """
    col = 1
    for nro,color in rul:
        if color == 'V':
            console.print (f"[bold white on green]{nro:^12}[/]", end="")
            print()
        elif color == 'R':
            console.print (f"[bold white on red]{nro:^4}[/]", end="")
            col += 1
        else:
            console.print (f"[bold white on black]{nro:^4}[/]", end="")
            col += 1

        if col == 4:
            print()
            col = 1

def imprimir_tirada (nro : int, color : str):
    if color == 'V':
        console.print (f"Obtuvimos [bold white on green]{nro}{color}[/]")
    elif color == 'R':
        console.print (f"Obtuvimos [bold white on red]{nro}{color}[/]")
    else:
        console.print (f"Obtuvimos [bold white on black]{nro}{color}[/]")

def simulacion(ruleta: list[(int,str)]):
    """Realiza una simulación y obtiene indicadores sobre la misma

    :param ruleta: Ruleta generada, para determinar los colores de los numeros
    :type ruleta: lista de tuplas (numero,color)
    """
```

```
    cantidad_pares = cantidad_impares = 0
    cantidad_rojos = cantidad_negros = 0
    tiradas_primera_docena = tiradas_segunda_docena = tiradas_tercera_docena = 0
    cantidad_ceros = 0
    jugadas = 1000

    # Generar 1000 numeros enteros aleatorios en el rango de la ruleta (0-36)
    for i in range(jugadas):
        tirada_nro = random.randint(0,36)
        tirada_color = ruleta[tirada_nro][1]
        #imprimir_tirada(tirada_nro,tirada_color)

        if tirada_nro%2 == 0:
            cantidad_pares += 1
        else:
            cantidad_impares += 1

        if tirada_color == 'R':
            cantidad_rojos += 1
        elif tirada_color == 'N':
            cantidad_negros += 1
        else:
            cantidad_ceros += 1

        docena = tirada_nro // 12
        if docena == 1:
            tiradas_primera_docena += 1
        elif docena == 2:
            tiradas_segunda_docena += 1
        else:
            tiradas_tercera_docena += 1
    porcentaje_ceros = cantidad_ceros*100/jugadas

    console.print()
    table = Table(title="Resultados de la simulación")
    table.add_column("Resultado")
    table.add_column("Valor",justify="center")
    table.add_row("Cantidad de pares",str(cantidad_pares))
    table.add_row("Cantidad de impares",str(cantidad_impares))
    table.add_row("Cantidad de tiradas 1° docena",str(tiradas_primera_docena))
    table.add_row("Cantidad de tiradas 2° docena",str(tiradas_segunda_docena))
    table.add_row("Cantidad de tiradas 3° docena",str(tiradas_tercera_docena))
    table.add_row("Porcentaje de ceros",str(porcentaje_ceros)+"%")
    table.add_row("Cantidad de rojos","[bold white on red]"+str(cantidad_rojos))
    table.add_row("Cantidad de negros","[bold white on black]"+str(cantidad_negros))
    console.print(table)

def principal():
    ruleta = generar_ruleta()
    imprimir_ruleta(ruleta)
    simulacion(ruleta)

if __name__ == "__main__":
    principal()
```

Capítulo 3

Archivos

En muchos casos, los programas requieren almacenar datos de forma persistente, de manera que estos datos se mantengan entre distintas ejecuciones del programa.

Los archivos son el mecanismo más básico y ampliamente utilizado en cualquier lenguaje de programación para lograr esta persistencia de datos.

Los archivos difieren significativamente de las bases de datos en el sentido de que son universales, lo que significa que cualquier programa que comprenda el formato de los datos almacenados en un archivo puede leerlos e incluso escribir sobre ellos, sin importar qué programa los haya generado previamente.

Existen notables diferencias entre los archivos y las bases de datos. Las bases de datos ofrecen ventajas significativas en términos de velocidad y capacidad de manipulación de grandes cantidades de datos. Además, proporcionan características de seguridad, como mecanismos de autenticación y autorización, para gestionar datos sensibles. Otro aspecto destacado de las bases de datos es su capacidad para mantener la consistencia de los datos, evitando la presencia de información incoherente o inconsistente.

A pesar de estas ventajas, la manipulación de datos a través de archivos ha sido ampliamente utilizada históricamente para transmitir información entre sistemas heterogéneos. Esta práctica es común cuando se requiere transferir datos de un programa a otro que no puede comunicarse directamente entre sí. Los archivos permiten que un programa almacene datos en un archivo y que otro programa, incluso escrito en un lenguaje o sistema diferente, pueda manipular esos datos. Este mecanismo ha sido tradicionalmente utilizado y sigue siendo utilizado en el campo de la ciencia de datos, donde se requiere la transferencia y manipulación conjunta de datos provenientes de diferentes fuentes.

3.1. Tipos de archivos

Cuando se requieren manipular datos almacenados en archivos, es fundamental considerar el formato o tipo de archivo. Existen dos tipos principales de archivos: archivos de texto y archivos binarios.

Los archivos de texto contienen información representada como una secuencia de caracteres legibles, escritos en una determinada codificación (UTF-8, ASCII, etc.). Estos archivos utilizan caracteres especiales, como el retorno de carro o la nueva línea, para delimitar cada línea y permitir la manipulación individual de cada dato. El formato de archivo de texto es adecuado para almacenar datos legibles por humanos, como texto plano, configuraciones o registros de datos en formato tabular.

Por otro lado, los archivos binarios no dependen de los caracteres de retorno de carro para delimitar los datos. En cambio, requieren de algún mecanismo específico para indicar la forma en que los datos están organizados dentro del archivo.Los archivos binarios son archivos en los que la información se almacena en un formato no legible directamente por los humanos y pueden contener datos de diversos tipos, como números, imágenes, audio, video, entre otros, donde la representación en formato de texto no sería práctica o eficiente. A diferencia de los archivos de texto, los archivos binarios no están codificados como caracteres legibles, sino que utilizan una representación binaria de los datos.

En resumen, los archivos de texto son adecuados para almacenar datos en formato de texto legible, mientras que los archivos binarios son utilizados para guardar datos más complejos y diversos, sin requerir la conversión a cadenas de texto.

3.2. Apertura y cierre

Para interactuar con un archivo en Python, es necesario informar al sistema operativo qué archivo se pretende utilizar, dónde está ubicado y qué operaciones se van realizar en él. Para lograr esto, se debe realizar la operación de apertura del archivo, indicando al sistema operativo que el programa está utilizando ese archivo y, potencialmente, evitando que otros programas accedan a él simultáneamente. Al finalizar el uso del archivo, es importante cerrarlo.

El cierre de un archivo cumple dos propósitos básicos. En primer lugar, notifica al sistema operativo que ya no se está utilizando y, por lo tanto, otros programas pueden acceder a él. En segundo lugar, en el caso de operaciones de escritura de datos en el archivo, el cierre garantiza que las últimas escrituras se almacenen correctamente en el disco o en el medio de almacenamiento utilizado. Al realizar escrituras, es probable que la biblioteca de manipulación de archivos o el propio sistema operativo espere a acumular varias operaciones de escritura antes de almacenarlas en el dispositivo de almacenamiento. Si un archivo no se cierra correctamente, es posible que las últimas escrituras no se envíen al destino, por lo que es importante cerrar siempre los archivos.

Python nos proporciona dos funciones para estas operaciones, llamadas precisamente `open` y `close`. La función open recibe dos parámetros. El primero es el nombre del archivo, una cadena que contiene el nombre y la extensión del archivo, y opcionalmente la ubicación utilizando rutas relativas o absolutas para especificar el directorio donde se encuentra el archivo. El segundo parámetro es el modo de apertura, donde se indica el tipo de archivo (texto o binario) y la operación que deseamos realizar en dicho archivo.

Open devuelve una estructura de datos que nos permite operar con el archivo. Esta estructura cuenta con un método llamado close, que se utiliza para indicar que el archivo ha sido cerrado correctamente.

En general, es recomendable mantener los archivos abiertos el menor tiempo posible. Por lo tanto, se debe abrir el archivo, realizar las operaciones necesarias y cerrarlo inmediatamente. No debe dejarse un archivo abierto más tiempo del necesario, especialmente durante el ingreso de datos por parte del usuario.

3.3. Modos de apertura

El modo de apertura que se especifica como parámetro en la función open es una cadena de texto que puede contener una o dos letras. Estas letras indican la operación que se desea realizar en el archivo. Para la operación de lectura, se utiliza la letra “R” (read), mientras que para la operación de escritura se utiliza la letra “W” (write).

El modo de apertura “W” tiene la particularidad de que, si el archivo en el que se desea escribir no existe, se crea automáticamente. Sin embargo, si el archivo ya existe, su contenido se borra por completo y se empieza a escribir desde cero, es decir, como un archivo vacío.

Existe otro modo de apertura para escritura denominado “X”. Este modo permite escribir datos en un archivo, pero si el archivo ya existe, impide borrar su contenido original y arroja un error. En resumen, el modo “X” es utilizado para escribir en un archivo completamente nuevo sin sobrescribir uno existente.

Otro modo de escritura es el modo “A” (append), que significa agregar. Cuando se abre un archivo en modo “A”, si el archivo no existe, se crea. Sin embargo, si el archivo ya existe, el contenido previo no se borra, sino que se comienza a escribir a continuación del último dato almacenado en el archivo.

Por último, existe el modo “R+” que permite tanto la lectura como la escritura en un archivo. Al abrir un archivo en modo “R+”, todas las operaciones de lectura y escritura se realizan desde el principio del archivo, aunque con operaciones de desplazamiento se puede indicar una posición específica para continuar a partir de ella.

Es importante mencionar que también se debe indicar el tipo de archivo con una letra “T” para archivos de texto y una letra “B” para archivos binarios. Si no se especifican estas letras, el modo de apertura predeterminado es de lectura de archivos de texto (“RT”).

Modo	Descripción	Acción sobre el archivo
`r`	Lectura	Abre el archivo para lectura (modo texto)
`w`	Escritura	Abre el archivo para escritura (modo texto)
`x`	Escritura exclusiva	Abre el archivo para escritura exclusiva (modo texto)
`a`	Adjuntar	Abre el archivo para agregar contenido al final (modo texto)
`r+`	Lectura y escritura	Abre el archivo para lectura y escritura (modo texto)
`rb`	Lectura binaria	Abre el archivo para lectura (modo binario)
`wb`	Escritura binaria	Abre el archivo para escritura (modo binario)
`xb`	Escritura exclusiva binaria	Abre el archivo para escritura exclusiva (modo binario)
`ab`	Adjuntar binario	Abre el archivo para agregar contenido al final (modo binario)
`r+b`	Lectura y escritura binaria	Abre el archivo para lectura y escritura (modo binario)

3.4. Lectura de archivos de texto

Para leer archivos de texto, hay tres métodos disponibles en la estructura devuelta por la función open. El primer método es `read`, que lee todo el contenido del archivo y devuelve una variable de tipo string. Este método es adecuado para archivos de texto relativamente pequeños, ya que carga todo el contenido en una sola variable. La principal desventaja es que, al leer todo el contenido, es responsabilidad del programador manipular el formato si el archivo tiene una estructura interna. Por ejemplo, si el archivo está separado por comas, será responsabilidad del programa dividir los datos según los delimitadores. Si el archivo contiene varias líneas de texto, read las lee todas y devuelve una cadena que incluye los caracteres de nueva línea o retorno de carro que indican el final de cada línea.

El segundo método es `readline`, que lee una sola línea y deja el archivo en un estado en el que la siguiente llamada leerá la línea que se encuentre a continación. De esta manera, readline devuelve una cadena con el contenido de una línea y permite la lectura en un ciclo para procesar líneas sucesivas. Este es el método más adecuado cuando se trata de archivos relativamente grandes, ya que al leer las líneas una a una, no se carga todo el contenido del archivo en la memoria, sino que se mantiene una única línea en memoria a la vez.

Por último, el método `readlines` (en plural) recorre todo el archivo y lee cada una de las líneas, devolviendo una lista donde cada posición contiene una línea de texto. Si se lo utiliza en un archivo con, por ejemplo, 20 líneas, el mismo devolverá una lista con 20 elementos, cada uno representando una línea del archivo en el orden en que fueron leídas. Readlines se encarga de dividir el archivo según los caracteres de nueva línea, pero ocupa más memoria, ya que todo el contenido del archivo se almacena en la lista. Por lo tanto, no sería apropiado para leer archivos de texto demasiado grandes, en el orden de varios megabytes, ya que consumiría una cantidad considerable de memoria.

Por lo tanto, para leer todo el contenido de un archivo de texto en una variable e imprimirla, se puede programar el siguiente bloque:

```
archivo = open("datos.txt")
contenido = archivo.read()
print(contenido)
archivo.close()
```

Y para leer un archivo más largo linea a linea:

```
archivo = open("datos.txt")

linea = archivo.readline()
while linea:
    print(linea)
    linea = archivo.readline()

archivo.close()
```

En este ejemplo, se abre el archivo “datos.txt” en modo lectura. Luego se utiliza un ciclo while para leer cada línea del archivo utilizando el método readline(). Dentro del ciclo, se imprime la línea y se lee la siguiente línea. El ciclo continúa hasta que se alcanza el final del archivo, es decir, cuando el método readline() devuelve una cadena vacía. Por último, se cierra el archivo utilizando el método close().

3.5. Escritura de archivos de texto

Para la escritura de archivos de texto existen dos métodos principales. El primer método es `write`, que permite escribir una cadena en el archivo. Al utilizar este método, la cadena se escribe en el archivo sin incluir automáticamente caracteres de nueva línea. Si se necesita escribir varias líneas de texto, es responsabilidad del programador agregar los caracteres de nueva línea en los lugares adecuados dentro de la cadena.

El segundo método es `writelines`, que recibe una lista de cadenas y escribe cada una de ellas en el archivo de texto en el orden en que aparecen en la lista. Al igual que el método write,writelines no agrega automáticamente caracteres de nueva línea, por lo que es responsabilidad del programador asegurarse de que las cadenas de la lista incluyan los caracteres de nueva línea donde sea necesario.

Por ejemplo:

```
nombres = ['Juan', 'María', 'Carlos', 'Laura']

archivo = open("nombres.txt", "w")

for nombre in nombres:
    archivo.write(nombre + '\n')

archivo.close()
```

En este ejemplo, se crea una lista llamada `nombres` con algunos nombres. Luego se abre el archivo “nombres.txt” en modo escritura. A continuación, se utiliza un bucle for para recorrer cada elemento de la lista nombres. En cada iteración, se utiliza el método write() para escribir el nombre en el archivo, seguido de un carácter de nueva línea \\`n`. Esto garantiza que cada nombre se escriba en una línea nueva del archivo.

3.6. Ejercicios

3.6.1. Lectura de un archivo de números

Desarrollar un programa que lea todo el contenido del archivo numeros.txt que contiene múltiples líneas de texto, cada una de ellas con un número entero. Al leer el archivo se debe almacenar todos los números en una lista. A continuación el programa debe manipular los números cargados en la lista para:

- Calcular e imprimir el promedio de todos los números
- Calcular e imprimir la cantidad de números mayores al promedio
- Generar y mostrar una nueva lista que contenga todos los números pares

proceso_listas.py

```
def cantidad_menor(lista, techo):
    """
    Cuenta los elementos de una lista que sean menores a un tope
```

```
    :param lista: lista de valores
    :type lista: lista de datos que soporte comparación por menor
    :param techo: valor máximo para filtrar
    :type techo: el mismo de los elementos de la lista
    :returns: la cantidad de elementos cuyo valor sea menor al techo
    :rtype: entero
    """
    c = 0
    for x in lista:
        if x < techo:
            c += 1

    return c

def cantidad_mayor(lista, piso):
    """
    Cuenta los elementos de una lista que sean menores a un tope

    :param lista: lista de valores
    :type lista: lista de datos que soporte comparación por menor
    :param techo: valor máximo para filtrar
    :type techo: el mismo de los elementos de la lista
    :returns: la cantidad de elementos cuyo valor sea menor al techo
    :rtype: entero
    """
    c = 0
    for x in lista:
        if x > piso:
            c += 1

    return c


def promedio(lista):
    """
    Calcula el promedio simple de todos los elemento de una lista

    :param lista: lista
    :type lista: lista de números
    :returns: el promedio de todos los valores de la lista
              o 0 si la lista está vacía
    :rtype: flotante
    """
    cantidad = len(lista)
    if cantidad == 0:
        return 0

    suma = 0
    for x in lista:
        suma += x

    return suma / cantidad


def existe(lista, buscado):
    """
    Busca un valor en una lista e informa si lo pudo encontrar

    :param lista: una lista de cualquier tipo
    :param buscado: el valor que se busca
    :tipo buscado: el mismo tipo que el de los datos de la lista
    :returns: verdadero si el elemento buscado existe en la lista
              y falso en caso contrario
    :rtype: boolean
    """
    for x in lista:
        if x == buscado:
            return True

    return False
```

lectura_numeros.py

```
from proceso_listas import *

def leer_archivo(nombre_archivo):

    numeros = []

    archivo = open(nombre_archivo)

    while True:
        linea = archivo.readline()
        if not linea:
            break
        n = int(linea[:-1])
        numeros.append(n)

    archivo.close()

    return numeros


if __name__ == "__main__":
    lista = leer_archivo("numeros.txt")
    promedio_todos = promedio(lista)
    mayores = cantidad_mayor(lista, promedio_todos)
    pares = [x for x in lista if x % 2 == 0]

    print("Del archivo se leyeron los números")
    print(f"El promedio de esos números es de {promedio_todos:5.2f}")
    print(f"Y {mayores} son mayores que el promedio")
    print("Listado de los pares:")
    for x in pares:
        print(x)
```

3.6.2. Lectura de un csv con códigos postales

El archivo cp.csv contiene el listado de los códigos postales de tres provincias argentinas. En cada línea se encuentran, separados por un símbolo de punto y coma (;) los siguientes datos:

- Provincia: representada por una letra mayúscula según el estándar ISO correspondiente.
- Código: es un número entero de cuatro dígitos que identifica una localidad o varias localidades vecinas. Puede estar repetido.
- Nombre: es el nombre de la localidad correspondiente a ese código.

Se requiere leer todo el contenido del archivo y guardarlo en una lista que contenga un elemento por cada línea. En cada elemento debe almacenarse alguna estructura de datos que permita acceder individualmente a cada dato que conforma un codigo postal.

Luego de la carga el programa debe permitir el ingreso de uno o más códigos numéricos y listar la provincia y nombre de todas las localidades asignadas a dichos códigos

cp.csv (primeras líneas)

```
K;4743;ACONQUIJA
K;4701;ACOSTILLA
K;4234;ACHALCO
K;5263;ADOLFO E. CARRANZA
K;4139;AGUA AMARILLA (LA HOYADA, DPTO. SANTA MARIA)
```

lectura_cp.py

```
def leer_codigos(nombre_archivo):

    codigos = []
    archivo = open(nombre_archivo)

    for linea in archivo.readlines():
        codigo = tuple(linea[:-1].split(";"))
        codigos.append(codigo)

    archivo.close()

    return codigos


def buscar_codigo(lista, buscado):
    encontrados = []

    for c in lista:
        if c[1] == buscado:
            encontrados.append(c)

    return encontrados
    # Con comprensión de listas
    # return [c for c in lista if c[1] == buscado]

if __name__ == "__main__":
    codigos = leer_codigos("cp.csv")

    buscado = input("Ingrese un código a buscar (fin con cadena vacía):")
    while buscado:
        encontrados = buscar_codigo(codigos, buscado)
        if encontrados:
            for c in encontrados:
                print(f"{c[0]}: {c[2]}")
        else:
            print("No se encontró ninguna localidad")
        buscado = input("Ingrese un código a buscar (fin con cadena vacía):")
```

3.6.3. Análisis del archivo de propinas

El archivo tips.csv contiene los datos de las propinas recibidas por los empleados de una pizzería en New York. El archivo contiene los siguientes campos separados por comas

- total_bill: Importe total del servicio
- tip: Importe de la propina
- sex: Sexo del cliente
- smoker: El cliente es fumador (Yes, No)
- day: Día (Juev, Vie, Sab, Dom)
- time: Horario (Almuerzo, Cena)
- size: Tamaño de la mesa (comensales)

Se requiere desarrollar un programa que desde los datos del archivo informe:

- ¿Quién paga más propinas?¿Hombres o mujeres?
- ¿Cuáles son los días más 'lentos', con menos propinas?
- ¿Cuál es el promedio de las propinas?
- ¿Cuándo se vendió la orden más grande (qué día y en qué turno)?

tips.csv (primeras líneas)

```
total_bill,tip,sex,smoker,day,time,size
16.99,1.01,Mujer,No,Dom,Cena,2
10.34,1.66,Hombre,No,Dom,Cena,3
21.01,3.5,Hombre,No,Dom,Cena,3
23.68,3.31,Hombre,No,Dom,Cena,2
```

tips.py

```
propinas_hombres = propinas_mujeres = 0
propina_fumador = propina_no_fumador = 0
propinas_acum = [0]*4
promedio = 0
cant_operaciones = 0
dia_orden_mayor = None
turno_orden_mayor = None
monto_orden_mayor = None

########################################################################
# Para utilizar la librería mejorada de consola (rich) primero hay que
# instalarla utilizando el comando pip install rich
########################################################################
from rich import print
from rich.console import Console
from rich.panel import Panel

c = Console()
c.clear()

first_row = True
with open("tips.csv","r") as file:
    for fila in file:
        # Debo saltear el primer registro ya que contiene los titulos
        if first_row:
            # Indicar que ya no estoy en la primera fila
            first_row = False
        else:
            cant_operaciones += 1
            # Hacer todo el procesamiento
            # Ahora debo separar la fila leída en sus componentes
            campos = fila[:-1].split(",")
            # print(campos)
            # Saco los campos que necesito
            total = float(campos[0])
            propina = float(campos[1])
            sexo = campos[2]
            dia = campos[4]
            horario = campos[5]

            # Acumular por separado las propinas de hombres y mujeres
            if sexo == 'Hombre':
                propinas_hombres += propina
            else:
                propinas_mujeres += propina

            # Obtener los datos de la mayor orden
            # Cuándo? Cuando sea la primera o sea mayor a la anterior
            if monto_orden_mayor == None or total > monto_orden_mayor:
                monto_orden_mayor = total
                dia_orden_mayor = dia
                turno_orden_mayor = horario

            # Solo trabajo 4 días así que puedo acumular las propinas
            # en un vector de 4 elementos, relacionados
            # a Jue, Vie, Sab, Dom
            if dia == 'Juev':
                propinas_acum[0] += propina
            elif dia == 'Vie':
                propinas_acum[1] += propina
            elif dia == 'Sab':
                propinas_acum[2] += propina
            else:
                propinas_acum[3] += propina

# Calcular la propina promedio
promedio = (propinas_hombres+propinas_mujeres)/cant_operaciones

# Recorrer el vector de propinas y ver cual es el menor
menor_indice = 0  # Voy a suponer que es el primero
menor_propina = propinas_acum[0]
menor_dia = ""

for i in range(len(propinas_acum)):
    if propinas_acum[i] < menor_propina:
```

```
        menor_propina = propinas_acum[i]
        menor_indice = i
# Cuando termine tengo que ver a qué dia corresponde ese índice
if menor_indice == 0:
    menor_dia = "Jueves"
elif menor_indice == 1:
    menor_dia = "Viernes"
elif menor_indice == 2:
    menor_dia = "Sábado"
else:
    menor_dia = "Domingo"

c.print(Panel("Análisis de propinas en New York"))

print (f"[white bold on green]Propinas de los hombres:[/] {propinas_hombres:.2f}")
print (f"[white bold on green]Propinas de las mujeres:[/] {propinas_mujeres:.2f}")
print (f"[white bold on green]Propina promedio:[/] {promedio:.2f}")
print (f"La mayor venta fué de [on green]{monto_orden_mayor}[/] y se realizó un [on green]{dia_orden_mayor}[/] en
↪  el turno [on green]{turno_orden_mayor}[/]")
print (f"Los días más lentos son los [on green]{menor_dia}[/] con solamente [on green]{menor_propina}[/] de
↪  propina")
c.print(Panel("Fin de la ejecución"))
```

Capítulo 4

Colecciones avanzadas

4.1. Conjuntos

4.1.1. Introducción

Los conjuntos son una estructura de datos proporcionada por el lenguaje que permite almacenar y manipular colecciones de elementos. A diferencia de las listas o las tuplas, los conjuntos tienen características distintivas que los hacen únicos.

Una de las principales características de los conjuntos es su capacidad para eliminar automáticamente elementos duplicados. Esto significa que, al insertar un elemento repetido en un conjunto, este será ignorado y no se agregará nuevamente.

Otra diferencia importante es que los conjuntos no admiten acceso indexado. No es posible acceder a elementos individuales utilizando el operador de corchetes o rebanadas, como se puede hacer en las listas o tuplas.

Sin embargo, la característica principal de los conjuntos es su eficiencia en las operaciones de búsqueda. Mientras que en las listas o tuplas las búsquedas pueden ser lentas, ya que requieren recorrer todos los elementos, en los conjuntos se utiliza una estrategia de dispersión que agiliza significativamente esta tarea. La estructura de datos desordena los elementos de manera inteligente, permitiendo localizar rápidamente un elemento en base a su criterio de dispersión, sin necesidad de recorrer todos los elementos.

Esta propiedad hace que los conjuntos sean especialmente útiles para realizar búsquedas de elementos sin duplicados. Además, los conjuntos son ideales para realizar operaciones de conjuntos matemáticos, como uniones, intersecciones y diferencias. Estas operaciones, que requerirían una programación relativamente compleja y lenta con listas o tuplas, pueden ser realizadas de manera eficiente y sencilla con conjuntos.

En resumen, los conjuntos son una valiosa estructura de datos que permiten almacenar y manipular colecciones de elementos sin duplicados. Su eficiencia en las operaciones de búsqueda y su capacidad para realizar operaciones de conjuntos de manera rápida los convierten en una herramienta poderosa para resolver diversos problemas.

4.1.2. Operaciones de datos

Creación

Existen tres formas de crear conjuntos en Python. La primera opción es utilizar la función `set()`, que crea un conjunto vacío. Este conjunto vacío se puede asignar a una variable para su posterior manipulación. Una vez creado el conjunto vacío, se pueden utilizar métodos para insertar elementos en él.

La segunda opción es crear un conjunto mediante la enumeración de elementos conocidos. Esto se asemeja a la notación matemática, donde se definen los elementos del conjunto entre llaves y separados por comas. Por ejemplo: `{1, 2, 3}`.

La tercera opción es utilizar comprensiones de conjuntos, que son similares a las comprensiones de listas. Se utiliza la misma sintaxis de comprensión, pero se delimita con un par de llaves en lugar de un par de corchetes. En ambos casos, es importante tener en cuenta que los conjuntos no permiten elementos duplicados, por lo que la cantidad de elementos en el conjunto resultante puede ser menor a la cantidad de elementos especificados en la creación. Los elementos duplicados serán ignorados. Por ejemplo: `{random.randint(1,10000) for x in range(100)}` crea un conjunto con hasta 100 números al azar sin repeticiones.

Inserción

Una vez que se ha creado un conjunto utilizando una de las tres alternativas mencionadas anteriormente, es posible agregar nuevos valores al conjunto utilizando dos métodos específicos.

El primer método es `add()`, que permite agregar un nuevo elemento al conjunto verificando que no exista previamente. En caso de que el elemento ya esté presente en el conjunto, el método add() no generará ninguna acción ni devolverá ningún error. El conjunto permanecerá sin modificaciones. Sin embargo, si el elemento que se intenta agregar no se encuentra en el conjunto, el método add() lo añadirá correctamente, expandiendo el conjunto con un elemento adicional.

El segundo método es `update()`, el cual recibe como parámetro una secuencia iterable, como un rango, una cadena, una tupla, una lista o incluso otro conjunto. Este método se encarga de recorrer todos los elementos de la secuencia iterable y los inserta en el conjunto original. Sin embargo, al igual que el método add(), el método update() garantiza que solo se insertarán elementos que no estén duplicados y que no existan previamente en el conjunto. Esto significa que no se agregarán elementos repetidos al conjunto durante el proceso de actualización.

Eliminación

Además de agregar elementos a un conjunto, también es posible eliminar elementos utilizando varios métodos disponibles.

El primer método es `remove()`, el cual recibe como parámetro el valor que se desea eliminar del conjunto. Este método elimina el elemento especificado y, en caso de que el elemento no exista en el conjunto, generará un error. Es importante tener cuidado al utilizar remove() para asegurarse de que el elemento a eliminar realmente esté presente en el conjunto.

El segundo método es `discard()`, que también recibe como parámetro el valor que se desea eliminar. Si el elemento existe en el conjunto, discard() lo eliminará sin generar errores. Sin embargo, si el elemento no está presente, discard() simplemente ignorará la solicitud y permitirá que el programa continúe sin interrupciones.

Otra opción para eliminar elementos es el método `pop()`, el cual selecciona y elimina arbitrariamente un elemento del conjunto. El elemento eliminado se devuelve como resultado de la llamada a pop(), lo que permite al usuario conocer qué elemento fue eliminado en caso de necesitarlo.

Por último, el método `clear()` borra todos los elementos del conjunto, dejándolo vacío. Este método es útil cuando se desea reiniciar el conjunto y eliminar todos sus elementos sin eliminar la variable en sí.

Recorridos

Cuando se trabaja con conjuntos y se desea recorrer sus elementos, se puede utilizar la instrucción for de manera similar a como se recorren las listas. Con la sintaxis for variable in conjunto, se realizará una iteración por cada elemento almacenado en el conjunto, asignando cada valor a la variable del ciclo for.

Es importante tener en cuenta que al recorrer un conjunto, los elementos no se encontrarán en un orden específico. A diferencia de las listas, donde los elementos se mantienen en el orden en que se insertaron, el recorrido de un conjunto mostrará los elementos completamente mezclados y desordenados. Esto se debe a que los conjuntos utilizan una estrategia de dispersión para lograr búsquedas rápidas, lo cual implica que los datos se almacenan de manera dispersa y desordenada.

Si se requiere recorrer un conjunto en un orden específico, como un orden numérico o alfabético, no es posible lograrlo directamente con un conjunto. Sin embargo, los conjuntos disponen del método `sorted()`, el cual devuelve una nueva lista con los mismos elementos del conjunto, pero ordenados según sus valores. Es importante destacar que esto implica la creación de una nueva estructura de datos (la lista ordenada) y no afecta el orden interno del conjunto.

A veces, al recorrer un conjunto, puede parecer que los elementos están ordenados, especialmente si se insertan solo números enteros. Sin embargo, esto no se puede garantizar para otros tipos de datos o para conjuntos más grandes. Por lo tanto, nunca se debe confiar en que un conjunto tendrá los datos ordenados, a menos que se utilice el método sorted() para obtener una lista ordenada explícitamente.

4.1.3. Operaciones de conjuntos

Los conjuntos son especialmente útiles para eliminar elementos duplicados, ya que garantizan la unicidad de los datos almacenados. Además, permiten realizar operaciones de conjuntos similares a las que se utilizan en matemáticas.

Pertenencia

La operación principal en un conjunto es verificar la pertenencia de un elemento. Para esto, se utiliza el operador `in`, que verifica si un valor dado pertenece al conjunto. El operador in devuelve True si el elemento está presente en el conjunto y False en caso contrario. La diferencia principal con respecto a las listas es que la velocidad de búsqueda con el operador in no es proporcional a la cantidad de elementos almacenados en el conjunto. Incluso si el conjunto es muy grande, la verificación de pertenencia se realizará de manera eficiente.

Cuando se trabaja con más de un conjunto, se pueden realizar operaciones de conjuntos, como la unión, la intersección y la diferencia. Estas operaciones se aplican utilizando diferentes métodos.

Unión

La unión de conjuntos se puede lograr mediante el método `union()`. Este método recibe como parámetro un segundo conjunto y devuelve un nuevo conjunto que contiene todos los elementos no repetidos de ambos conjuntos. También está disponible el método update(), que realiza la unión modificando el conjunto original.

Intersección

La intersección de conjuntos se puede lograr utilizando el método `intersection()`. Este método recibe como parámetro un segundo conjunto y devuelve un nuevo conjunto con los valores que existen en ambos conjuntos. Además, el método `intersection_update()` realiza la intersección modificando el conjunto original.

Diferencias

La diferencia de conjuntos se puede obtener mediante el método `difference()`. Este método se aplica a un conjunto y recibe como parámetro otro conjunto. Retorna un nuevo conjunto con los elementos presentes en el primer conjunto pero no en el segundo. También está disponible el método `difference_update()`, que realiza la diferencia modificando el conjunto original.

La diferencia simétrica, también conocida como diferencia exclusiva, se obtiene utilizando el método `symmetric_difference()`. Este método recibe como parámetro un segundo conjunto y devuelve un nuevo conjunto que contiene todos los elementos presentes en ambos conjuntos, excepto aquellos que existen en ambos conjuntos al mismo tiempo. El método `symmetric_difference_update()` realiza la diferencia simétrica modificando el conjunto original.

4.1.4. Casos de aplicación

4.1.5. Eliminación de repetidos de una lista

En este ejemplo se dispone de una lista llamada `lista` que contiene elementos duplicados. Para eliminar los duplicados, se convierte la lista en un conjunto utilizando la función `set()`. Los conjuntos en Python sólo pueden contener elementos únicos, por lo que al convertir la lista en un conjunto, automáticamente se eliminan los elementos duplicados.

Luego se convierte nuevamente el conjunto a una lista utilizando la función `list()`y se asigna en una nueva lista llamada `lista_sin_duplicados` que contiene los elementos de la lista original sin duplicados.

Finalmente, se imprime la lista sin duplicados para verificar el resultado.

```
# Lista con elementos duplicados
lista = [1, 2, 3, 4, 2, 3, 5, 6, 1, 7, 8, 5, 9]

# Convertir la lista en un conjunto
# para eliminar duplicados
conjunto = set(lista)

# Convertir el conjunto nuevamente a una lista
lista_sin_duplicados = list(conjunto)

# Imprimir la lista sin duplicados
print(lista_sin_duplicados)

#[1, 2, 3, 4, 5, 6, 7, 8, 9]
```

4.1.6. Operaciones entre múltiples conjuntos

En este ejemplo se dispone de dos conjuntos `curso_a` y `curso_b` que representan los nombres de personas inscriptas en dos cursos diferentes y sobre ellos se utilizan las siguientes operaciones de conjuntos:

La operación de unión (`union()`) combina ambos conjuntos y devuelve un nuevo conjunto con todos los elementos únicos de ambos conjuntos.

La operación de intersección (`intersection()`) encuentra los elementos comunes entre ambos conjuntos y devuelve un nuevo conjunto con esos elementos.

La operación de diferencia (`difference()`) encuentra los elementos que están en el primer conjunto pero no en el segundo conjunto y devuelve un nuevo conjunto con esos elementos.

La operación de diferencia simétrica (`symmetric_difference()`) encuentra los elementos que están en uno de los conjuntos pero no en ambos conjuntos y devuelve un nuevo conjunto con esos elementos.

Cada operación se aplica a los conjuntos y el resultado se almacena en una variable correspondiente. A continuación se imprimen los conjuntos resultantes para ver los nombres de las personas según cada operación.

```
# Conjunto de nombres de personas del curso A
curso_a = {"Juan", "María", "Pedro", "Luisa", "Ana"}

# Conjunto de nombres de personas del curso B
curso_b = {"Pedro", "Ana", "Sofía", "Carlos"}

# Unión: personas inscriptas en al menos uno de los cursos
union = curso_a.union(curso_b)
print(union)
# {'Juan', 'Carlos', 'María', 'Pedro', 'Luisa', 'Ana', 'Sofía'}

# Intersección: personas inscriptas en ambos cursos
interseccion = curso_a.intersection(curso_b)
print(interseccion)
# {'Ana', 'Pedro'}

# Diferencia: inscriptos sólo en el curso A
diferencia = curso_a.difference(curso_b)
print(, diferencia)
# {'María', 'Luisa', 'Juan'}

# Diferencia simétrica: inscriptos en
# uno de los cursos, pero no en ambos
diferencia_simetrica = curso_a.symmetric_difference(curso_b)
print(diferencia_simetrica)
# {'Carlos', 'Sofía', 'María', 'Juan', 'Luisa'}
```

4.2. Diccionarios

4.2.1. Introducción

Los diccionarios son una estructura de datos altamente versátil y poderosa que permite almacenar información de manera distinta a otras estructuras disponibles en el lenguaje. A diferencia de estas, los diccionarios no almacenan conjuntos o grandes volúmenes de datos, sino que se basan en pares de datos conocidos como “clave” y “valor”. También denominados “arreglos asociativos” en otros lenguajes, los diccionarios establecen una relación entre una clave única en todo el conjunto y un valor asociado a esa clave.

En esencia, los diccionarios almacenan pares de datos, donde la clave actúa como identificador único para un dato específico, y el valor corresponde al dato asociado a esa clave. Este concepto es similar al de las claves primarias utilizadas en estructuras de bases de datos relacionales. Una característica destacada de los diccionarios es la velocidad excepcional de las operaciones de búsqueda. Estas operaciones son tan rápidas como en los conjuntos, lo que implica que la velocidad de búsqueda se mantiene constante, independientemente de la cantidad de datos almacenados. En un diccionario, incluso con una gran cantidad de datos, las búsquedas se realizan a la misma velocidad que si hubiera pocos datos.

Al igual que los conjuntos, los diccionarios no tienen un índice que establezca un orden o posición específica para cada dato y no permiten claves duplicadas. Esta característica es fundamental para su funcionalidad y eficiencia en la recuperación de datos asociados a claves específicas.

Los diccionarios son especialmente útiles en situaciones en las que es necesario asociar un dato con un valor identificador. Por ejemplo, al almacenar datos de nombres de personas, es posible utilizar un diccionario en el que

los nombres estén asociados con los números de documento de cada individuo. De esta manera, al proporcionar el número de documento al diccionario, se obtendría el nombre correspondiente asociado a ese documento.

Si bien es posible lograr una funcionalidad similar utilizando arreglos y utilizando el valor clave como índice del arreglo, en muchos casos el valor clave no puede ser equiparado a un índice de arreglo. Por ejemplo, si se requiere almacenar datos de personas y utilizar el número de documento como índice de un arreglo, se enfrentaría a la impracticabilidad de crear un arreglo con 100 millones de elementos para cubrir todas las posibilidades de números de documento de 8 dígitos. Los diccionarios, en cambio, permiten utilizar cualquier tipo de dato disponible en Python como clave, ya sean valores numéricos, cadenas u otros tipos de datos.

La flexibilidad de los diccionarios radica en su capacidad para establecer relaciones directas entre datos y valores identificadores, independientemente de su tipo o dominio. Esto los convierte en una herramienta poderosa para abordar una amplia gama de situaciones donde es necesario asociar y recuperar datos de manera eficiente.

4.2.2. Operaciones

Creación

La sintaxis utilizada para crear diccionarios es bastante similar a la de los conjuntos. Existen tres mecanismos principales para crear un diccionario.

El primero de ellos es utilizando la función `dict()`, la cual inicializa un diccionario vacío al que se le pueden ir agregando pares clave-valor.

Otra forma de crear un diccionario es mediante la enumeración explícita de sus datos. Se utiliza un par de llaves ({}) y se enumeran los valores separados por comas. Sin embargo, a diferencia de los conjuntos, en los diccionarios se debe establecer un par clave-valor para cada valor enumerado. Esto se realiza utilizando dos puntos (:) para asociar la clave con su respectivo valor.

Por último, se puede utilizar la comprensión de diccionarios para crear diccionarios de manera concisa. La sintaxis es similar a la comprensión de conjuntos, utilizando un par de llaves que encierra una expresión de comprensión. En este caso, se establece el par clave-valor, donde tanto la clave como los valores pueden depender de los valores iterados en un ciclo for, utilizando dos puntos (:) para asociarla con el valor correspondiente.

Acceso

Una vez que se ha creado un diccionario, es posible acceder a sus valores utilizando el operador de corchetes ([]), que permite el acceso indexado similar al de los arreglos. En este caso, se indica la clave correspondiente al valor que se desea obtener. A diferencia de los arreglos, en los diccionarios la clave puede ser de cualquier tipo válido en Python. Por lo tanto, se puede utilizar el nombre del diccionario seguido del operador de acceso indexado, y dentro de éste se especifica la clave que se busca. El acceso indexado devuelve en tiempo constante el valor asociado con la clave. Si la clave no existe en el diccionario, se devuelve el valor vacío `None`.

Otra forma de acceder a los valores almacenados en el diccionario es utilizando el método `get()`. Este método recibe como parámetro la clave que se busca y, en caso de que la clave no se encuentre en el diccionario, devuelve None por defecto. Sin embargo, get() acepta un segundo parámetro opcional que permite especificar un valor por defecto. De esta manera, se puede acceder al valor asociado a la clave sin necesidad de utilizar una instrucción condicional, y se obtendrá el valor por defecto si la clave no existe en el diccionario.

Por último, existe un método llamado `setdefault()`, que recibe una clave y un valor por defecto como parámetros. Si la clave existe en el diccionario, el método devuelve el valor asociado a esa clave. En caso contrario, crea una nueva entrada en el diccionario asociando la clave con el valor por defecto especificado en el segundo parámetro. Este método es útil para evitar la verificación de existencia de la clave y asignar un valor predeterminado si no se encuentra en el diccionario.

Para poder conocer la longitud de un diccionario se dispone de la instrucción `len` que funciona de la misma manera que con todas las colecciones e iterables.

Inserción

Para agregar nuevos elementos o modificar elementos existentes en un diccionario, se utiliza el operador de acceso indexado de manera similar a como se haría con un arreglo. Es decir, se utiliza el nombre del diccionario seguido del símbolo de corchetes con la clave deseada en su interior, y luego se utiliza el operador de asignación (=) para asignar un nuevo valor a esa clave.

Si la clave ya existe en el diccionario, el valor asociado a esa clave se actualizará con el nuevo valor asignado. En caso de que la clave no exista previamente en el diccionario, se creará una nueva entrada con la clave y el

valor especificados. De esta manera, se puede agregar o modificar elementos en el diccionario de forma dinámica y flexible.

Borrado

Para eliminar elementos de un diccionario, existen dos métodos principales. El primero es utilizar la instrucción ‘del‘, seguida del nombre del diccionario y entre corchetes la clave que se desea borrar. Esta instrucción simplemente elimina la clave y el valor asociado a ella.

El segundo método es el uso del método ‘pop()‘. Este método recibe como argumento la clave que se desea eliminar y devuelve el valor asociado a esa clave. Además, elimina la clave y su valor del diccionario. Si se proporciona un segundo argumento opcional al método ‘pop()‘, se especifica un valor por defecto que será retornado si la clave no existe en el diccionario. Esto evita que se produzca un error en caso de que la clave no exista.

Otra opción es utilizar el método ‘clear()‘, el cual borra todos los pares clave-valor del diccionario, dejándolo vacío como si estuviera recién creado. Este método no requiere ningún argumento.

Finalmente, la instrucción ‘del‘ también puede utilizarse para eliminar todo el diccionario por completo, incluyendo sus valores contenidos, lo cual implica eliminar también la variable del diccionario en sí.

Estas opciones permiten la manipulación y eliminación de elementos en un diccionario de acuerdo a las necesidades específicas del programa.

Recorrido

Los diccionarios ofrecen tres formas de ser recorridos. El método keys() retorna un conjunto con todas las claves, el cual puede ser recorrido y manipulado con operaciones de conjuntos. Por otro lado el método values() retorna una lista con todos los valores. Corresponde que sea una lista porque si buen las claves no pueden repetirse, los valores sí pueden hacerlo.

Tambien pueden recorrerse los pares ordenados (clave, valor) con un ciclo for. Pero en este caso el ciclo posee dos variables que se asignan en cada vuelta. Para ello se dispone del método items() que retorna una secuencia de tuplas. En cada vuelta del for puede asignarse la tupla o directamente desempaquetar la misma en dos variables. En el ejemplo siguiente se las desempaqueta en las variables numero y nombre:

```
diccionario = {1: "Lunes", 2: "Martes", 3: "Miércoles"}

for numero, nombre in diccionario.items():
    print(f"El día {numero} se llama {nombre}")
```

Capítulo 6

Programación funcional

6.1. Introducción

Al desarrollar software se identifican paradigmas de programación que gobiernan el enfoque con el que se encara el problema, se diseña una solución y finalmente se redacta el código fuente. Entre los paradigmas más utilizados se destacan el paradigma estructurado y el paradigma orientado a objetos.

Otro paradigma de uso habitual es el paradigma funcional. El mismo tiene una base matemática bastante compleja y por tal motivo existen pocos lenguajes denominados puros, es decir, que permiten únicamente desarrollar con este paradigma. Sin embargo, actualmente todos los lenguajes de uso cotidiano (tales como Java, C#, Python o Javascript) incluyen ciertos elementos de esta metodología.

A pesar de que estos lenguajes están mayormente concentrados en los paradigmas estructurado y orientado a objetos, ofrecen algunas características de programación funcional. Esto se logra mediante la posibilidad de trabajar con un nuevo tipo de dato denominado “función”.

El origen de esta perspectiva puede parecer desconcertante, pero se la puede analizar desde el punto de vista de las variables. Cuando se utiliza la programación estructurada, los programas poseen variables donde se almacenan valores. En esta modalidad, cada variable tiene la capacidad de almacenar un único valor de alguno de los tipos de datos ofrecidos por el lenguaje, tales como números, cadenas, booleanos, etc. En cambio, en la programación orientada a objetos, las variables pueden contener objetos, los cuales pueden tener múltiples atributos y métodos. Aunque es válido argumentar que las variables solo contienen referencias a objetos, en esencia, sigue siendo cierto que cada variable almacena un objeto.

En la programación funcional, las variables almacenan código fuente, es decir, conjuntos de instrucciones del lenguaje. Por lo tanto, si una variable se declara como una función, es posible ejecutar el código que esta contiene. Y dado que son variables pueden cambiar su contenido, por lo tanto, si en algún momento se asigna un código diferente a la variable y se ejecuta su contenido nuevamente, se llevará a cabo una ejecución distinta. En conclusión, siempre se ejecutará el conjunto de instrucciones que esté almacenado en la variable en ese momento específico.

6.2. Funciones de orden superior

Los lenguajes que ofrecen programación funcional, incluso en su forma más básica, brindan la posibilidad de tener variables cuyo contenido sea una función. También se pueden tener parámetros de tipo función, lo cual implica que un método puede ejecutar una función sin necesidad de saber con exactitud su comportamiento.

De la misma manera, una función o un método de una clase puede retornar una función para que sea almacenada desde la llamada al mismo.

Los métodos que poseen parámetros o retorno de tipo función se denominan métodos o funciones de orden superior. Esta noción tiene una gran relevancia matemática y es muy útil en diversas situaciones de programación.

6.3. Variables de tipo función

Una variable puede almacenar una referencia a una función. En el siguiente ejemplo se declara una variable llamada `imprimir` y se asigna su valor con el nombre de la función `print`. Debe notarse que no se utilizan los paréntesis que efectivamente iniciarían una llamada a la función print. Luego de la asignación, la variable puede ser ejecutada y el resultado es que se ejecuta un print.

```
imprimir = print

imprimir("Hola, mundo!")
```

6.4. Colecciones de funciones

De la misma manera que una variable puede almacenar una referencia a una función, se pueden crear colecciones cuyos elementos sean funciones.

En el siguiente ejemplo se crea una calculadora básica con las cuatro operaciones elementales de la aritmética implementadas con una función por cada una. Para que el usuario pueda seleccionar la operación deseada se ofrece un menú con un número natural por cada opción. Al momento de ejecutar la función correspondiente se accede a una lista que contiene en cada posición una referencia a la función correspondiente en el menú. Esto permite acceder a la lista usando como índice la opción seleccionada, y dado que el valor retornado es efectivamente una función se la puede ejecutar pasándole los parámetros que la misma requira. Para ello la lista `operaciones` es inicializada con un valor `None` ya que no hay ningúna operación con número 0 en el menú, y luego cada una de las referencias a las funciones de las operaciones aritméticas.

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

def resta(a, b):
    return a - b

def multiplicacion(a, b):
    return a * b

def division(a, b):
    return a / b

def ejecutar_calculadora():
    operaciones = [None, suma, resta, multiplicacion, division]
    print("Calculadora básica")
    print("Seleccione una operación:")
    print("1. Suma")
    print("2. Resta")
    print("3. Multiplicación")
    print("4. División")
    print("0. Salir")

    opcion = int(input())

    while opcion ≠ 0:
        if opcion ≥ 1 and opcion ≤ 4:
            a = float(input("Ingrese el primer número: "))
            b = float(input("Ingrese el segundo número: "))

            resultado = operaciones[opcion](a, b)
            print("El resultado es:", resultado)
        else:
            print("Opción inválida. Intente nuevamente.")

        print()
        print("Seleccione una operación:")
        opcion = int(input())

ejecutar_calculadora()
```

6.5. Funciones lambda

Las funciones lambda, también conocidas como funciones anónimas, son funciones pequeñas y concisas que se definen sin un nombre y se utilizan en el momento en que se necesitan. A diferencia de las funciones normales, que se definen utilizando la palabra clave `def`, las funciones lambda se crean utilizando la palabra clave `lambda`.

La sintaxis básica de una función lambda es la siguiente:

```
lambda argumentos: expresión
```

Donde lambda es la palabra clave que indica que se está creando una función lambda, la lista de argumentos son los parámetros de la función, separados por comas pero sin los paréntesis que exige el bloque def y la expresión es la que se evalúa y devuelve como resultado de la función.

Una función lambda está limitada a una sola expresión y no puede contener múltiples líneas de código. En el caso de necesitar realizar múltiples operaciones, ejecutar varias líneas de código o usar estructuras de control, debe desarrollarse una función regular con def.

En el ejemplo anterior de la calculadora se pueden evitar todas las definiciones de las funciones suma, resta, multiplicacion y división de la siguiente manera:

```
suma = lambda a, b: a + b
resta = lambda a, b: a - b
multiplicacion = lambda a, b: a * b
division = lambda a, b: a / b
operaciones = [None, suma, resta, multiplicacion, division]
```

6.6. Secuencias y operaciones funcionales

Existen funciones de orden superior que simplifican la manipulación de los datos de las secuencias.

6.6.1. Filter

La función `filter()` recorre una secuencia y ejecuta por cada uno de los datos de la misma una función recibida como parámetro y que se espera que devuelva un valor de verdad, tanto valores booleanos como convertibles a boolean (truthy). Esta función retorna una nueva secuencia cuyos valores son los mismos de la secuencia original pero únicamente aquellos para los que la función retorna un valor verdadero.

La sintaxis de filter es:

```
filter(funcion, secuencia)
```

Dado que filter retorna una nueva secuencia, la misma debe ser recorrida con una instrucción for o convertida a una colección, con las funciones `list()` o `set()`.

Por ejemplo, la siguiente llamada a filter extrae los números pares de una lista:

```
es_par = lambda x: x \% 2 == 0
numeros = [2,1,35,8,5,21,4,6,9,6,3,2]
pares = list(filter(es_par, numeros))

print(pares)
#[2, 8, 4, 6, 6, 2]
```

6.6.2. Map

La función `map()` permite generar una nueva secuencia de forma tal que cada elemento se obtenga con el resultado de ejecutar una función a cada elemento de la secuencia original. La sintaxis es idéntica a la de la función filter, de forma tal que el primer parámetro debe ser la función a repetir y el segundo contiene la secuencia que debe procesarse.

En el siguiente ejemplo se obtienen una lista con los cuadrados de los numeros de la lista original:

```
numeros = [2,1,35,8,5,21,4,6,9,6,3,2]
cuadrados = list(map(lambda x: x ** 2, numeros))

print(cuadrados)
#[4, 1, 1225, 64, 25, 441, 16, 36, 81, 36, 9, 4]
```

6.6.3. Reduce

La función `reduce()` recorre una secuencia aplicando una función de pliegue o reducción, obteniendo como resultado un único valor. Para la operacion de reducción requiere una función de dos parámetros que realice alguna operación con los mismos y retorne un valor. Al ejecutarse toma los dos primeros elementos de la secuencia e invoca a la función recibida, la cual debe retornar un resultado. A continuación invoca nuevamente a dicha función enviándole como parámetros el tercer elemento de la secuencia y el resultado de la primera ejecución. Luego continúa invocando la función con cada uno de los restantes elemntos de la secuencia y el retorno de la ejecución anterior.

Por ejemplo, si se dispone de una función suma, que reciba dos números y retorne la suma aritmética de esos parámetros, puede utilizarse reduce de la siguiente manera:

```
numeros = [23,6,9]
total = reduce(suma, numeros)

print(total)
# 38
```

Al ejecutarse reduce, inicia invocando a la función suma con los dos primeros valores de la lista (23 y 6), obteniendo como resultado el número 29. A continuación vuelve a llamar a suma pasandole como parámetros 29 (el resultado anterior) y 9 (el tercer elemento de la lista), obteniendo 38. Y dado que la lista no posee más elementos retorna ese valor como resultado final de la reducción.

La función reduce fue movida al módulo `functools` por lo tanto para poder utilizarla debe ser importada con `from functools import reduce`.

Capítulo 7

Clases

7.1. Introducción

7.1.1. Programación orientada a objetos

La programación orientada a objetos (POO) es un paradigma de programación que organiza el código alrededor de objetos, los cuales son instancias de clases. En lugar de enfocarse en las acciones y funciones que se realizan en el programa, la POO se centra en la interacción entre los objetos, permitiendo modelar y representar de manera más precisa entidades del mundo real en el software.

En la POO, un objeto combina datos y funciones relacionadas en una sola estructura, lo que facilita su manipulación y comprensión. Los objetos se definen mediante clases, que actúan como plantillas o moldes para crear instancias específicas. Cada objeto tiene atributos (propiedades o características) que describen su estado y métodos (funciones o comportamientos) que definen su capacidad para realizar acciones y responder a eventos.

El enfoque orientado a objetos fomenta la reutilización de código y el diseño modular, ya que los objetos pueden ser utilizados en diferentes partes de un programa y en otros programas. Además, permite una abstracción más efectiva, ya que los objetos encapsulan tanto los datos como el comportamiento relacionado, lo que facilita el desarrollo y el mantenimiento del software a largo plazo.

La POO se basa en cuatro principios fundamentales:

Abstracción Permite representar características y comportamientos esenciales de un objeto en un modelo simplificado. Se identifican las propiedades (atributos) y acciones (métodos) relevantes de un objeto y se encapsulan en una clase.

Encapsulamiento Consiste en ocultar los detalles internos de un objeto y proporcionar una interfaz para interactuar con él. Los datos (atributos) y las funciones (métodos) de un objeto se encapsulan dentro de la clase, y solo se accede a ellos a través de métodos públicos definidos en la interfaz.

Herencia Permite crear nuevas clases basadas en clases existentes, aprovechando las características y comportamientos heredados. Una clase derivada (subclase) puede extender o especializar la funcionalidad de una clase base (superclase), heredando sus atributos y métodos, y agregando o modificando los necesarios.

Polimorfismo Permite que un objeto pueda presentar diferentes formas o comportamientos según el contexto en el que se utilice. Se puede invocar un método de diferentes objetos de una jerarquía de clases y obtener resultados diferentes, siempre que cumplan con la interfaz común definida en una clase base.

La programación orientada a objetos (POO) fue inventada con el objetivo de proporcionar una mejor forma de organizar y diseñar software complejo. Antes de la POO, los programas se estructuraban principalmente en torno a funciones y procedimientos, lo que a menudo resultaba en un código difícil de mantener, extender y reutilizar.

La POO surgió en la década de 1960 como una forma de abordar estos desafíos. Su objetivo principal era permitir una representación más fiel de los objetos del mundo real dentro del software, lo que facilitaría el desarrollo de aplicaciones más escalables y flexibles.

Al enfocarse en los objetos, la POO permitió una mejor modelización de los sistemas, ya que los objetos podían contener tanto datos (atributos) como comportamientos relacionados (métodos). Esto permitió que el código se estructurara de manera más intuitiva y modular, ya que los objetos podían interactuar entre sí mediante mensajes, encapsulando así su funcionalidad interna.

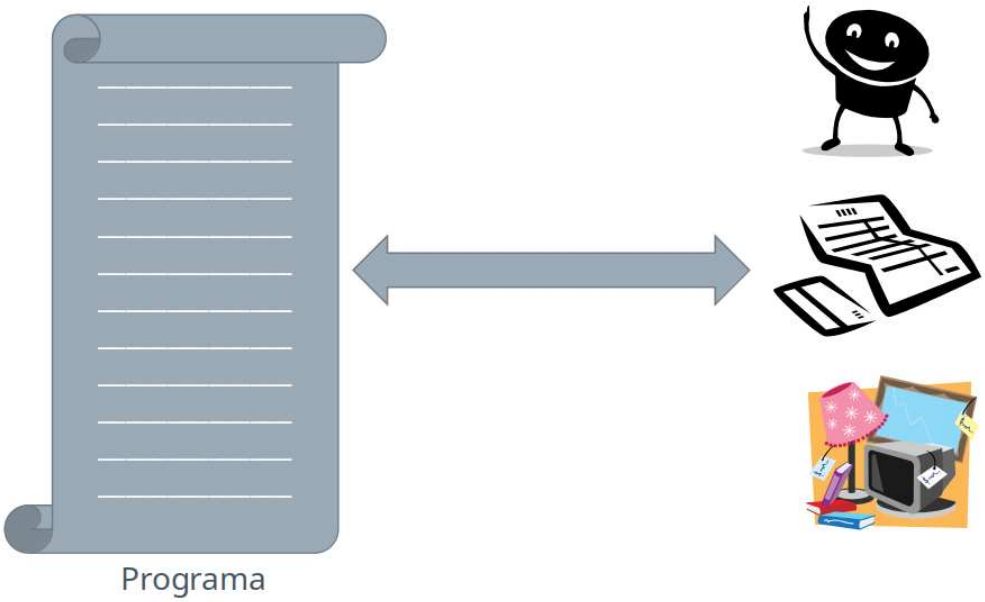


Figura 7.1: Programación estructurada

La programación orientada a objetos (POO) ofrece varias ventajas en comparación con la programación estructurada, especialmente en relación al concepto de la brecha semántica o semantic gap. La brecha semántica se refiere a la diferencia entre cómo se modela y se representa el mundo real en el software y cómo se percibe y comprende el mundo real por parte de los usuarios y desarrolladores.

De esta manera, si se requiere desarrollar un software que represente, por ejemplo, un sistema de ventas para un comercio, entre los conceptos principales detectados durante la etapa de análisis se encontrarían los de artículos, clientes y ventas. En la programación estructurada, el software sería redactado como una secuencia bastante larga de líneas de código, agrupadas y organizadas con algún criterio, pero orientado principalmente a la funcionalidad esperada por el software.

Sin embargo, todo software requiere ser susceptible a cambios luego de ser desarrollado. Tales cambios pueden provenir de diversos orígenes, tales como errores detectados durante el funcionamiento, requerimientos nuevos o incluso cambios externos, por ejemplo en la legislación vigente.

La programación orientada a objetos busca que el código fuente sea lo más parecido posible a la realidad, para que sea más simple identificar el punto donde debe ser modificado cuando se encuentre una modificación en los requerimientos, y que estos afecten en pocos lugares dentro de todo el código fuente. Así, si en el software del sistema de ventas, si hay un cambio en un requirimiento del proceso de ventas, los cambios en el código fuente estarían localizados principalmente en la clase que representa a la transacción de la venta, y no en la o las clases que representan a los artículos o a los clientes.

7.1.2. Clases y objetos

Cuando se busca resolver un problema utilizando la programación orientada a objetos, se aborda la representación del problema mediante la creación de objetos y las relaciones entre ellos. Cada objeto encapsula su propio estado y comportamiento, lo que permite modelar de manera más precisa las entidades y relaciones del dominio del problema.

Cada objeto tiene atributos que describen sus características y métodos que definen sus comportamientos o acciones asociadas. Muchos objetos comparten características que los definen, pudiendo ser agrupados o clasificados bajo una misma definición. Por ejemplo, dos objetos que realicen compras en un comercio generalmente tienen el mismo conjunto de datos y realizan las mismas acciones. A ambos se los puede clasificar como “Clientes”.

Una clase es una plantilla que define las características comunes compartidas por un conjunto de objetos para los que se aplica la misma definición. En esencia, una clase actúa como un modelo o plano para la creación de objetos. Contiene la estructura y el comportamiento que se aplicarán a cada instancia de esa clase.

Cuando se crea un objeto en la programación orientada a objetos, se basa en una clase específica. Cada objeto es una instancia individual y concreta de esa clase. Esto significa que cada objeto posee sus propios valores únicos para los atributos definidos en la clase, así como su propia capacidad para ejecutar los métodos asociados a esa

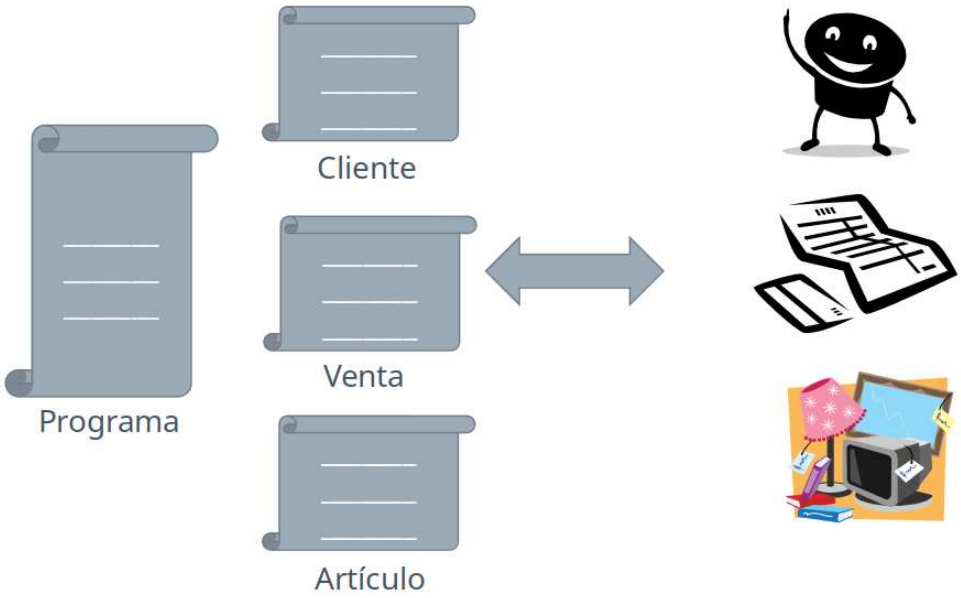


Figura 7.2: Programación orientada a objetos

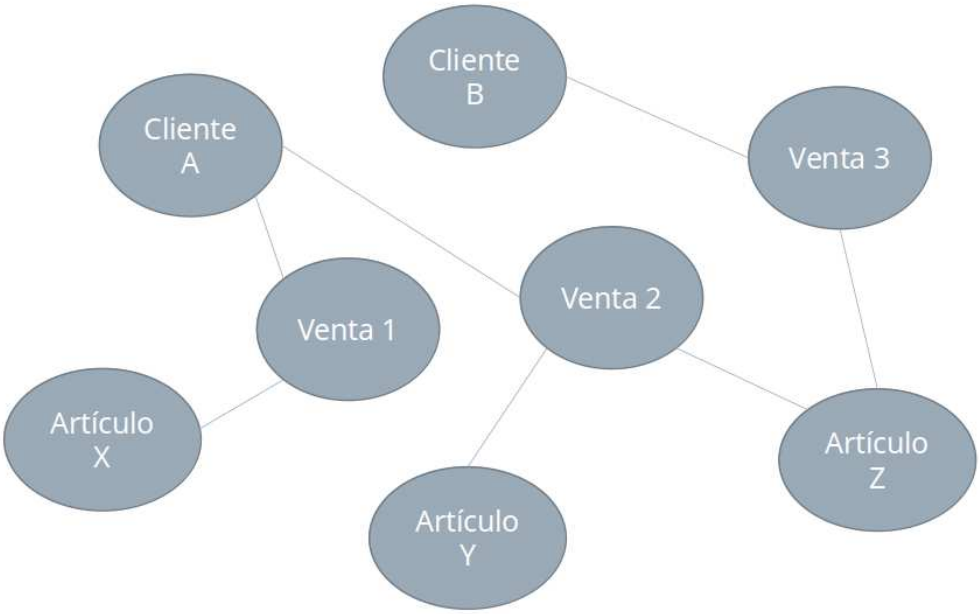


Figura 7.3: Objetos y sus relaciones

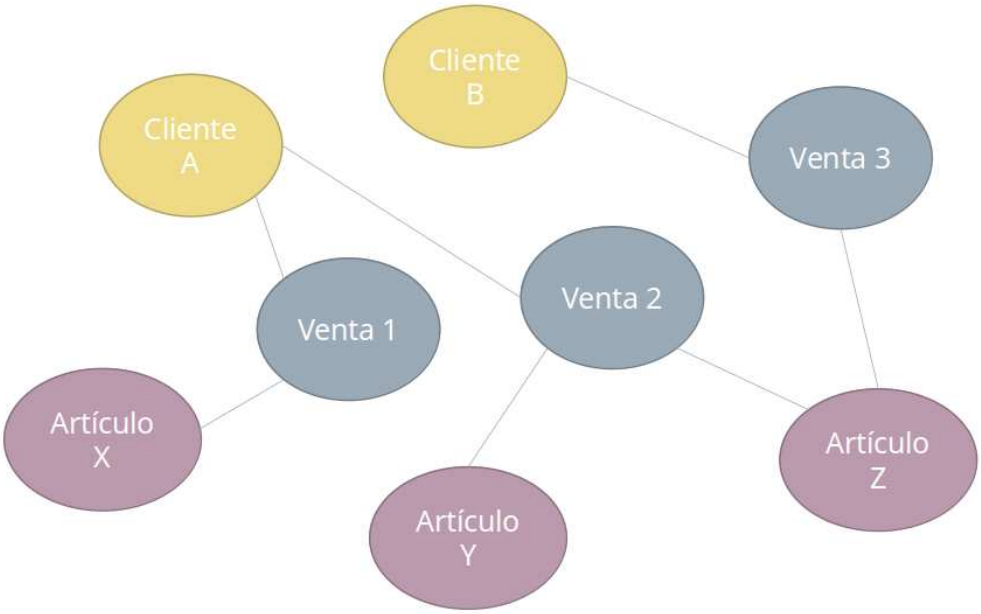


Figura 7.4: Objetos clasificados según su rol en el dominio

clase. Cada objeto es una entidad única y distinta que comparte las características y el comportamiento definidos en su clase, pero puede tener valores de atributos diferentes de las otras instancias de la misma clase y puede responder de manera independiente a los mensajes que reciba.

7.2. Clases

Para la creación de una clase se utiliza el bloque `class`, cuya sintaxis es:

```
class NombreClase:
    definición de métodos
```

El nombre de la clase generalmente consiste de un sustantivo en singular, con notación Pascal, es decir que esta commpuesto por letras en minúscula con su inicial en mayúscula. En caso de que el identificador esté formado por más de una palabra, se las diferencia sin un delimitador y con mayúscula en la primera letra de cada palabra, por ejemplo: CodigoPostal.

7.3. Atributos

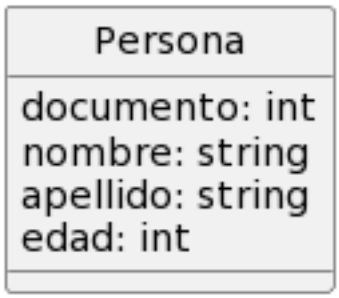
Los atributos no son declarados con una construcción gramatical en particular, simplemente se los asigna en el momento en que se los requiere crear dentro de algún método. Para diferenciar los atributos de las variables locales se los diferencia usando siempre la sintaxis `objeto.atributo`, indicando al objeto con el identificador seleccionado para referenciar al receptor del mensaje.

7.4. Constructor

El método constructor se denomina `__init__` y para funcionar require recibir como primer parámetro formal una variable que referencia al objeto que se está construyendo. El identificador de este parámetro puede ser cualquier nombre válido de variable, pero la convención prefiere denominarlo usualmente `self`.

El uso más habitual del método constructor es el de asignar valores iniciales para los atributos. Tales valores pueden ser recibidos como parámetros o asignados en forma arbitraria. Si bien las clases sólo pueden poseer un único constructor, se le pueden asignar valores por defecto a los parámetros para ofrecer que en la construcción se indiquen valores concretos durante la creación o se utilicen los valores por defecto.

Suponiendo una clase Persona con atributos para almacenar documento, nombre, apellido y edad de una persona, se presentan a continuación el diagrama de clases y el bloque class correspondiente:




```
class Persona:
    def __init__(self, documento=0, nombre="No", apellido="No", edad=0):
        self.documento = documento
        self.nombre = nombre
        self.apellido = apellido
        self.edad = edad
```

Para instanciar objetos de esta clase Persona se debe ejecutar el constructor con una sintaxis que puede parecer extraña: en lugar de invocar en forma explícita a la función __init__, se indica el nombre de la clase (en este caso Persona) seguido de un par de paréntesis en el que se pueden enviar parámetros. Y hay más, ya que el primer parámetro formal (self) no debe ser enviado, el mismo va a llegar en forma automática al método constructor, referenciando al objeto que se está creando.

Los métodos cuyos nombres inician y terminan con dos guiones bajos (__) se denominan **métodos mágicos** ya que no son invocados en forma explícita con su nombres sino que se ejecutan en forma automática a partir de otras operaciones. En otro capítulo se los analiza con mayor detalle.

Por lo tanto, para obtener instancias de personas se las puede crear de la siguiente manera: requiere el siguiente bloque de código:

```
# Los atributos se inicializan con
# valores por defecto
a = Persona()

# Los atributose se inicializan con
# los valores indicados en los
# parámetros actuales
b = Persona(1234, "Juan", "Perez", 23)

# Los atributos documento y apellido
# se inicializan con los valores enviados
# y los otros con valores por defecto
c = Persona(documento=1234, apellido="Castro")
```

Las clases pueden estar programadas en cualquier archivo de código fuente y en cualquier parte del mismo (afuera de una función), sin embargo es usual ubicar cada clase en un archivo cuyo nombre coincida con el de la clase. De esta manera, la clase Persona podría estar programada en un archivo persona.py. Para utilizar la clase deberá ser importada, usualmente con una instrucción de la forma `from persona import *`.

7.5. Referencias

Las variables a las que se les asigna un objeto se crean con un tipo llamado object. Sin embargo, la idea de que la variable almacena un objeto, a pesar de ser cómoda, es incorrecta.

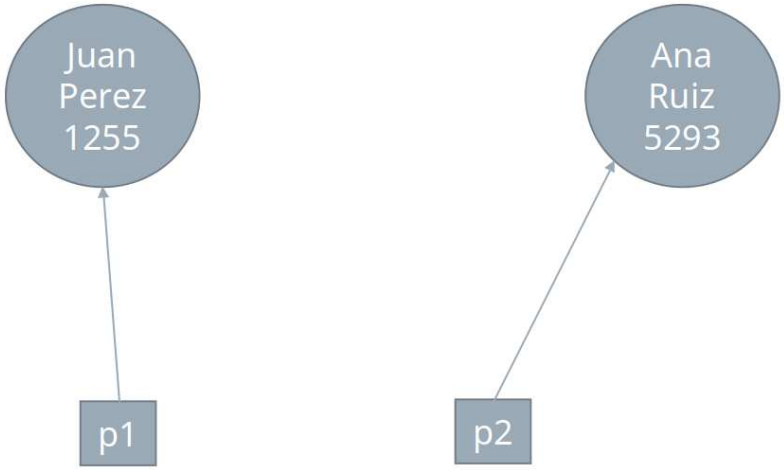
A diferencia de las variables de la programación estructurada, que son un almacenamiento que ocupa una porción de memoria con tamaño suficiente para que un dato entre completamente y que poseen un identificador, los objetos funcionan de forma diferente. La construcción de un objeto mediante una llamada a un método constructor efectivamente reserva memoria para almacenarlo, pero los objetos **no tienen nombre**!. Cada objeto posee una identidad provista durante su instanciación y que consiste en concreto en la dirección de memoria

donde esta almacenado. Pero sin un nombre no se les puede enviar mensajes y por lo tanto los objetos resultan inservibles.

Las variables de tipo object se denominan **referencias** y su valor efectivamente no es una instancia concreta de alguna clase. El dato que almacenan es la identidad de un objeto, es decir, la dirección de memoria donde el objeto se encuentra. Es práctico imaginar que una variable de tipo referencia guarda en realidad una flecha hasta el objeto.

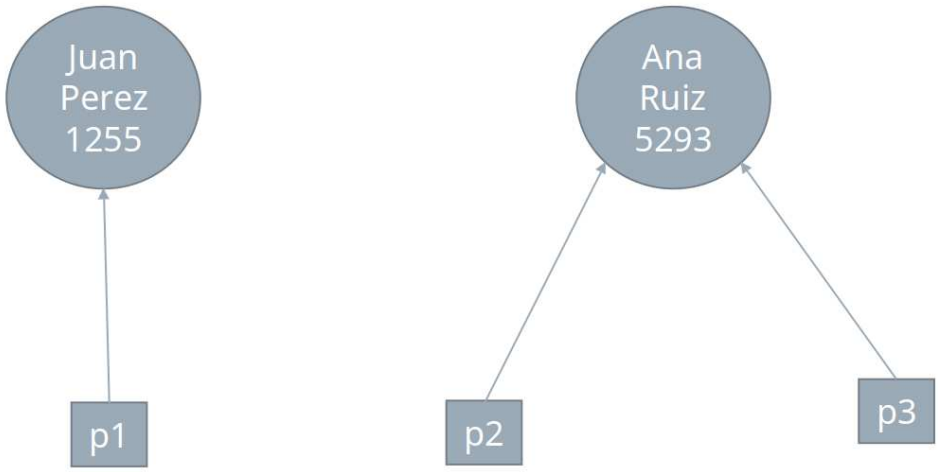
Por ejemplo, luego de ejecutar el siguiente código, los objetos quedan asignados como indica la figura:

```
p1 = Persona(documento=1255,nombre="Juan",apellido="Perez")
p2 = Persona(documento=5293,nombre="Ana",apellido="Ruiz")
```



Cada variable de tipo referencia puede señalar o apuntar a un único objeto, pero de la misma forma que dos variables pueden poseer el mismo valor, dos referencias pueden estar señalando al mismo objeto. En estos casos, el envío de mensajes desde una referencia o desde la otra afectan a la misma instancia concreta. La operación de asignación efectivamente copia los valores de las variables, y eso no es diferente en el caso de las referencias: si se asigna una referencia con el valor de otra, dado que ellas almacenan las “flechas”, ambas quedan referenciando al mismo objeto.

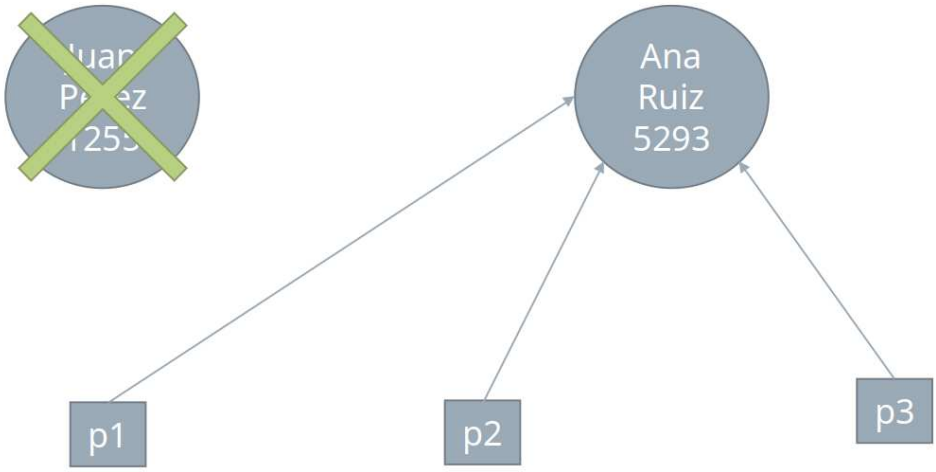
De esta manera, si a continuación del ejemplo anterior se ejecuta `p3 = p2`, el estado del programa queda como muestra la figura:



Otra consecuencia de esto es que la asignación no duplica objetos, la única forma de crear un objeto nuevo es mediante una llamada al constructor.

Finalmente, si un objeto creado deja de ser apuntado por alguna referencia, el mismo queda inaccesible y por lo tanto inservible, ya que no se le pueden enviar mensajes ni siquiera para consultar su estado. Eventualmente si hace falta memoria durante la ejecución del programa, el interprete de python podrá decidir eliminarlo de la memoria.

Si en el ejemplo se asigna `p1 = p2`, ninguna referencia queda apuntando a la persona Juan Perez:



7.6. Métodos

7.6.1. Introducción

La programación orientada a objetos propone la construcción de programas mediante la interacción entre objetos, los cuales son entidades que encapsulan tanto datos (atributos) como comportamientos (métodos) relacionados. En este enfoque, un programa se concibe como un conjunto de objetos que interactúan entre sí al intercambiar mensajes.

Cuando un objeto envía un mensaje a otro, está solicitando al objeto receptor que realice una acción o que proporcione información específica. Este proceso se denomina “paso de mensajes”. El objeto receptor, a su vez, atiende el mensaje y ejecuta el método correspondiente para responder adecuadamente a la solicitud.

La forma de implementar este mecanismo de mensajes en los lenguajes orientados a objetos es a través de lo que se denomina “métodos”. Los métodos son funciones asociadas a un objeto particular que definen su comportamiento y permiten que otros objetos interactúen con él. Los mismos pueden acceder a los atributos del objeto y realizar operaciones según la lógica definida en el código.

Los métodos que residen dentro de cada objeto y se ejecutan cuando el objeto recibe un mensaje. Los mensajes pueden contener parámetros que el método utilizará para llevar a cabo su operación, además de acceder a los atributos propios del objeto para actuar en consecuencia y enviar una respuesta como resultado del mensaje recibido.

En Python, los métodos se definen como funciones que reciben parámetros, pero el primer parámetro, por convención, es una referencia al propio objeto, que generalmente se nombra como `self`. Esta referencia self permite que el método acceda a los atributos y otros métodos del objeto actual, lo que facilita la manipulación de los datos y comportamientos específicos de ese objeto.

Además un objeto puede colaborar con otros objetos para llevar a cabo una tarea o cumplir con un objetivo específico. Cuando un objeto recibe un mensaje y atiende a dicho mensaje, puede requerir la colaboración de otros objetos para lograr su objetivo.

Para lograr esta colaboración, un método puede acceder a otros objetos y enviarles mensajes. Estos mensajes pueden incluir parámetros o información necesaria para que los objetos colaboradores realicen sus acciones. Luego, el método puede esperar las respuestas de estos objetos colaboradores y utilizar esa información para alcanzar sus propios objetivos y proporcionar una respuesta o resultado al mensaje original.

7.6.2. Sintaxis

Para agregar un método a una clase debe incluirse dentro del bloque `class` una definición de función con `def` cuyo primer parámetro sea `self`:

```
class Clase:

   def __init__(self, ... )
      # código del constructor

   def metodo(self, parámetros):
      # código del método
      return resultado
```

Así, en el contexto de una clase `Persona` con atributos `nombre` y `apellido`, es posible definir un método llamado `nombre_completo` que concatene ambos atributos y retorne el nombre completo de la persona. Este método se definiría dentro de la clase ”Personaz recibiría el parámetro ”self”, que es una referencia al objeto actual que recibe el mensaje de la siguiente manera:

```
class Persona:

    def nombre_completo(self):
        return f"{self.nombre} {self.apellido}"
```

Para enviar un mensaje de un objeto a otro, se utiliza la sintaxis de un punto seguido del nombre del método con paréntesis para enviar los parámetros necesarios. Esta sintaxis es equivalente a la de ejecutar una función, pero en este caso, está asociada a una referencia particular (objeto). En la lista de parámetros no se incluye al self, el mismo es llenado en forma implícita.

El operador punto es el que permite enviar el mensaje al objeto referenciado desde una variable de tipo referencia. Dado que las referencias pueden considerarse que almacenan una flecha que parte desde la variable y llega al objeto. Cuando se utiliza el operador punto en una referencia, este recorre la flecha hasta llegar al objeto al que está apuntando, y es a ese objeto al que se le envía el mensaje mediante la invocación del método.

```
per = Persona(123, "Juan", "Perez", 20)

print(per.nombre_completo())
```

Es importante tener en cuenta que si la referencia a la que se le envía el mensaje está vacía o tiene el valor `None` (nulo), se generará un error, ya que no hay un objeto apuntado al que pedirle nada. Por lo tanto, el programador debe garantizar que la referencia no esté vacía antes de enviar el mensaje.

Para evitar este tipo de errores, es común utilizar condicionales para verificar que la referencia no esté vacía antes de enviar el mensaje.

Capítulo 8

Encapsulamiento

8.1. Introducción

El encapsulamiento es un concepto fundamental en la programación orientada a objetos que se basa en la idea de que los objetos se comporten como “cápsulas”, es decir, como unidades cerradas que protegen su implementación interna. Esto significa que los detalles internos de la lógica, la naturaleza y la implementación de los atributos y métodos de un objeto están ocultos y no son accesibles directamente desde otros objetos. En lugar de permitir el acceso directo, los objetos deben comunicarse entre sí únicamente a través del paso de mensajes.

La finalidad del encapsulamiento es proteger y mantener la integridad y coherencia del objeto, evitando que otros objetos interfieran directamente con sus atributos o métodos internos. Esto promueve una mayor modularidad y flexibilidad en el diseño del software, ya que cada objeto puede cambiar su implementación interna sin afectar el funcionamiento de otros objetos que interactúan con él.

Al ocultar los detalles internos, el encapsulamiento mejora la abstracción y permite enfocarse en la interfaz pública del objeto, es decir, los métodos que están destinados a ser utilizados por otros objetos para interactuar con él. Esto crea una barrera protectora alrededor del objeto, lo que ayuda a prevenir errores y mantener la coherencia del sistema.

Para establecer una comunicación entre objetos, se utiliza el paso de mensajes, donde un objeto envía un mensaje a otro objeto solicitando la ejecución de un método específico o solicitando información. El objeto receptor responde a este mensaje atendiendo al método correspondiente y puede devolver el resultado o enviar un nuevo mensaje a otro objeto, lo que permite la colaboración y el trabajo conjunto entre los objetos del sistema.

8.2. Ventajas

8.2.1. Ocultamiento de la implementación

El encapsulamiento satisface la necesidad de permitir que un objeto pueda modificar su forma de funcionar sin afectar a los demás objetos con los que se comunica. Esta capacidad de adaptarse a cambios internos sin interrumpir la interacción con otros objetos es esencial para el mantenimiento y evolución del software a lo largo del tiempo.

Si se analiza por ejemplo un escenario donde un objeto tiene la responsabilidad de almacenar datos de manera persistente, podria hacerlo en un archivo de texto. Otros objetos interactúan con él enviándole mensajes para guardar o recuperar datos. En una primera versión del programa, esta puede ser la implementación utilizada, y los objetos que se comunican con él confían en que sus mensajes serán atendidos adecuadamente y que los datos se guardarán en un archivo.

Sin embargo, si en una versión futura el objeto decide cambiar su implementación para almacenar los datos en una base de datos o en una nube, en lugar de hacerlo en el archivo de texto, el objetivo es que este cambio no afecte al código existente de los otros objetos que interactúan con él.

Esto se logra a través del encapsulamiento. Los objetos que utilizan al objeto de almacenamiento no necesitan conocer los detalles internos de su implementación ni dónde se almacenan los datos. En su lugar, confían en la interfaz pública del objeto, es decir, los mensajes que pueden enviarle y las respuestas que pueden esperar.

De esta manera, los objetos que interactúan con el objeto de almacenamiento simplemente le envían mensajes como ”guardá este dato.º recuperá el dato almacenado”. No necesitan saber si el objeto utiliza un archivo de texto o una base de datos en la nube para almacenar los datos; simplemente confían en que sus mensajes serán procesados correctamente y que el objeto cumplirá con su responsabilidad de almacenamiento.

8.2.2. Validez del estado interno

El encapsulamiento surge también debido a la necesidad de mantener la validez del estado interno de un objeto. Un objeto requiere tener la tranquilidad de que sus atributos siempre poseen valores válidos acorde a su lógica interna. Si los atributos estuvieran expuestos y cualquier otro objeto pudiera modificarlos directamente, podrían asignarse valores que no cumplen con las restricciones lógicas del objeto. Como resultado, el objeto se vería forzado a incluir estructuras condicionales para verificar la validez de sus atributos cada vez que necesita operar con ellos. Esta situación generaría una complejidad innecesaria en el funcionamiento del objeto, requiriendo continuamente validaciones para garantizar la coherencia de los datos.

Entonces se recomienda ocultar los atributos de otros objetos y establecer que cualquier consulta o modificación de un atributo debe realizarse mediante el envío de mensajes específicos al objeto. De esta manera, el objeto tiene un control total sobre la validez de sus atributos y puede validarlos en el momento en que otro objeto intenta cambiarlos o asignarles un nuevo valor.

Por ejemplo, si un objeto representa un producto con un atributo de “precio”, que no puede ser negativo según la lógica del negocio, el encapsulamiento permite que cualquier intento de modificar el precio se realice mediante el envío de un mensaje específico al objeto, que incluya la nueva información. En este punto, el objeto puede verificar si el nuevo valor cumple con las reglas de validez o si es incorrecto. En caso de ser válido, el objeto procederá a asignar el nuevo valor y almacenar el dato correctamente. Por otro lado, si el valor no es válido, el objeto puede rechazar la asignación y tomar una decisión adecuada según el mecanismo que haya sido establecido, como lanzar una excepción, enviar un mensaje de error o tomar una acción alternativa.

Este enfoque permite al objeto garantizar que siempre tiene datos válidos y coherentes, ya que tiene el control total sobre la modificación de sus atributos. Asimismo, facilita el mantenimiento del código, ya que las validaciones y la lógica de negocio se concentran dentro del objeto, manteniendo una separación clara de responsabilidades entre objetos.

Se puede analizar esta característica a través de otro ejemplo. Cada persona tiene una billetera que contiene todos los billetes que posee en un momento dado. Cuando una persona necesita pedir prestado dinero a otra, debe enviarle un mensaje solicitando el préstamo. La persona receptora del mensaje conoce cuánto dinero tiene en su billetera y tiene el derecho de decidir si presta o no, ya que es su propio dinero. Puede responder afirmativamente, negativamente o incluso puede optar por no decir la verdad sobre la disponibilidad de dinero, aunque realmente lo tenga. Esta capacidad de decidir prestar o no es fundamental porque el dinero realmente le pertenece.

La importancia del encapsulamiento se hace evidente en este escenario. Si no existiera el encapsulamiento, es decir, si cualquier persona pudiera acceder directamente a la billetera de otra, podría tomar prestado dinero sin permiso y sin que el propietario se entere de la operación. Esto desencadenaría dos problemas. En primer lugar, el propietario de la billetera tendría menos dinero, lo que afectaría sus recursos.Pero el problema más significativo es que si el propietario no se diera cuenta de la reducción del dinero, podría tomar decisiones incorrectas al realizar compras o comprometerse con gastos sin saber que su dinero ha disminuido sin autorización.

Por lo tanto, para mantener la integridad de los datos y preservar la autonomía del objeto dueño de la billetera, se aplica el encapsulamiento. Los objetos ocultan la información sobre cuánto dinero tienen, de manera que no se pueda acceder directamente a su billetera desde otros objetos. La cantidad de dinero en la billetera solo será accesible si otro objeto envía un mensaje preguntando cuánto dinero tiene el propietario. Del mismo modo, si alguien solicita un préstamo, el objeto dueño del dinero decidirá si lo presta o no y nunca permitirá que su cantidad de dinero quede sin control.

8.2.3. Atributos calculados

Por otra parte, el encapsulamiento también brinda al objeto la capacidad de decidir si mostrar o no los datos almacenados en sus atributos cuando otros objetos los consultan. Esto permite que el objeto tenga el control sobre qué información se expone y cómo se presenta, según lo considere conveniente. Asimismo, el objeto puede optar por responder a las consultas calculando los datos a partir de otros atributos disponibles.

Cuando un objeto recibe un mensaje que consulta un dato específico, no es necesario que el objeto que envía el mensaje sepa si el dato que recibe es simplemente un atributo almacenado o si es calculado en el momento de la consulta. Este enfoque de abstracción oculta los detalles internos de la implementación y permite que el objeto proporcione la información solicitada de una manera coherente y transparente para el objeto solicitante.

Si por ejemplo un objeto “Círculo” tiene atributos como “radio” y “centro”, el mismo podría exponer métodos que simulen tener otros atributos, como “diámetro”, “superficie” y “circunferencia”. Estos atributos calculados podrían obtenerse en el momento de la consulta, basándose en los datos disponibles del círculo. Así una instancia de `Circulo` podría tener un método llamado `diametro()` que, en lugar de retornar el valor de un atributo, lo calcule

en función del radio disponible. Cuando otro objeto solicita el diámetro del círculo, simplemente accede a este método sin necesidad de saber si el diámetro es un atributo almacenado o si lo calcula en el momento.

8.3. Propiedades

Para implementar el encapsulamiento, una de las formas preferidas es utilizando el decorador `@property`. Este decorador permite programar los métodos de consulta y asignación, es decir, un método para obtener el valor de un atributo y otro método asociado para asignar un nuevo valor a ese mismo atributo. Cada uno de estos métodos, conocidos como getters (método de consulta) y setters (método de asignación), puede contener la lógica necesaria, tan sencilla o compleja como sea requerida.

Estos métodos se los decora especificando que conforman un par llamado **propiedad**. Una vez establecidos como propiedades, los objetos clientes pueden utilizar la misma sintaxis que utilizan para acceder a un atributo simple o para asignar un valor a un atributo. De esta manera, el objeto cliente sigue utilizando la sintaxis de atributos, pero en realidad, de forma implícita, se ejecutan los métodos de consulta y asignación correspondientes. Esto permite que el objeto reciba el mensaje de consultar o modificar un atributo sin necesidad de exponer directamente el acceso a sus atributos internos.

La sintaxis de estos métodos es bastante particular: el método de consulta se define con el nombre de la propiedad, generalmente un sustantivo, y se lo anota con @property. El método de asignación se anota con un decorador que lleva el nombre de la propiedad seguido de `.setter`, lo que indica que es el método setter asociado a esa propiedad. En el método setter, se recibe como parámetro self, como es habitual en todos los métodos, y el nuevo valor que se intenta asignar al atributo.

Por ejemplo, en la clase Persona del capítulo anterior, para que esté encapsulado el atributo `documento` se debería hacer:

```
class Persona:
    def __init__(self, documento, nombre, apellido, edad):
        self._documento = documento
        self.nombre = nombre
        self.apellido = apellido
        self.edad = edad


    @property
    def documento(self):
        return self._documento

    @documento.setter
    def documento(self, nuevo_documento):
        self._documento = nuevo_documento
```

Luego, cuando se intenta acceder al atributo documento en realidad se están ejecutando los métodos getter y setter:

```
p = Persona(11, "Juan", "Perez", 23)
p.documento = 22 #Ejecuta el método setter
print(p.documento) # Ejecuta el getter
```

8.4. Métodos mágicos

En Python, existen métodos que comúnmente se conocen como “métodos mágicos”. Aunque el término “mágicos”puede sonar confuso, estos métodos son efectivamente especiales y poseen características únicas que no se encuentran en otros lenguajes de programación, o que pueden variar de un lenguaje a otro.

Lo que distingue a estos métodos es que se invocan de manera implícita, es decir, no se llaman directamente utilizando la sintaxis tradicional de envío de mensajes o llamada a funciones. En lugar de eso, Python proporciona una forma especial de invocar estos métodos, mediante una convención de nomenclatura que los hace reconocibles. Estos métodos son conocidos como “Dunder methods”, una abreviatura de “double underscore methods”, debido a que sus nombres comienzan y terminan con dos guiones bajos (__).

Un ejemplo conocido de un método ”mágico.[es] `__init__`, que se utiliza como el constructor de una clase. Cuando se crea una instancia de una clase en Python, el método __init__ se llama de forma implícita para inicializar los

atributos y configurar el objeto. No es necesario invocarlo manualmente con su nombre ya que Python lo hace automáticamente al crear un objeto de la clase.

Un aspecto interesante de estos métodos mágicos es que permiten que las clases personalicen su comportamiento en ciertos contextos. Por ejemplo, el método `__str__` se utiliza para definir cómo se debe representar una instancia de una clase como una cadena de caracteres cuando se utiliza la función str(). Al definir este método en una clase, se puede controlar cómo se verá una instancia de esa clase cuando se imprima o se convierta en una cadena.

Estos métodos mágicos desempeñan un papel crucial en la programación orientada a objetos al proporcionar la capacidad de personalizar el comportamiento de las clases y los objetos creados a partir de ellas. Estos métodos son necesarios debido a la distinción fundamental entre los objetos creados a partir de una clase y los tipos de datos simples o primitivos que ofrece el lenguaje, como números, booleanos y cadenas.

En Python, los tipos de datos simples pueden participar en operaciones gracias a los operadores. Por ejemplo, los operadores de comparación permiten comparar dos números o dos cadenas, y las operaciones aritméticas aplicadas a números realizan cálculos aritméticos. Sin embargo, cuando se crean clases y se instancian objetos a partir de ellas, estos objetos no tienen la capacidad inherente de realizar operaciones entre ellos como la suma o la comparación de igualdad de sus valores.

Cuando se comparan por igualdad dos objetos, se obtiene simplemente un valor de verdad que indica si ambas referencias se refieren al mismo objeto o instancia. Sin embargo, no existe una forma natural de realizar comparaciones como “mayor que” o “menor que” entre objetos de una clase, ya que esta lógica no está definida de manera predeterminada para la mayoría de las clases personalizadas.

Con estos métodos se permite a los programadores agregar lógica específica a sus clases para habilitar operaciones personalizadas entre objetos de la misma clase. Por ejemplo, al definir los métodos `__lt__` (menor que) y `__gt__` (mayor que) en una clase, se puede personalizar cómo se deben comparar los objetos de esa clase en función de sus atributos internos. O, por ejemplo, con `__add__` para la suma, se consigue que los objetos realicen operaciones aritméticas personalizadas.

8.4.1. Método __str__

El método `__str__` es un componente fundamental en la programación orientada a objetos en Python. Este método se utiliza para proporcionar una representación en forma de cadena del estado interno de un objeto. Su lógica radica en mostrar, en formato de cadena, todos o algunos de los valores almacenados en los atributos del objeto. En otros lenguajes de programación, como Java o C#, este concepto se implementa mediante un método llamado toString, o el método asString de SmallTalk.

La ventaja de definir el método str con este nombre especial, con guiones bajos dobles (__str__), es que permite que Python realice conversiones implícitas del objeto a una cadena en diversas situaciones. Por ejemplo, cuando se utiliza la función print, esta puede recibir como parámetro una instancia de una clase definida por el usuario. En este caso, la función print invocará automáticamente al __str__ del objeto y mostrará por pantalla la representación que el objeto decide proporcionar de sí mismo en formato de cadena.

8.4.2. Métodos de comparación

La capacidad de comparar objetos de manera personalizada es esencial para la aplicación de lógicas específicas relacionadas con la igualdad, desigualdad y otros criterios de comparación.

Un ejemplo de estos métodos especiales es `__eq__`, que se utiliza para la comparación por igualdad. La lógica subyacente en este método es la de determinar si un objeto es igual a otro en función de los criterios definidos por el programador. Por ejemplo, en el contexto de una clase Persona, se lo podría programar para que devuelva True si todos los atributos de dos objetos Persona tienen el mismo valor. Esta lógica personalizada reflejaría la definición específica de igualdad para objetos de esta clase en el contexto de la aplicación.

Es importante enfatizar que Python no puede adivinar automáticamente cómo se deben comparar objetos personalizados. La razón detrás de esto es que la comparación entre objetos puede depender de lógicas de negocio específicas y variar ampliamente de una aplicación a otra. Por lo tanto los programadores deben proporcionar esta lógica de comparación personalizada en los métodos especiales correspondientes.

Además existen otros métodos especiales, como `__ne__` para la desigualdad, `__lt__` para “menor que”, `__gt__` para “mayor que”, `__le__` para “menor o igual que” y `__ge__` para “mayor o igual que”. Cada uno de estos métodos permite a los objetos de una clase personalizada participar en comparaciones utilizando los operadores correspondientes.

8.4.3. Métodos aritméticos

Considerando una clase que represente un concepto matemático, por ejemplo una Fracción, tiene sentido aplicar operaciones aritméticas sobre fracciones, tales como sumarlas o multiplicarlas. Un método especial que podría ser relevante en este contexto es `__add__`, que se utiliza para sobrecargar el operador de suma (+). Al programar este método dentro de la clase Fraccion, se le proporciona la capacidad de sumar dos instancias de fracciones. La lógica subyacente en el método determina cómo se realiza esta suma y qué significa en el contexto del dominio en el que se va a usar la clase.

A continuación se presenta una posible implementación de la clase Fraccion y de la sobrecarga de los operadores de adición y multiplicación:

```
from math import gcd

class Fraccion:
def __init__(self, numerador, denominador):
# Simplificamos la fracción al máximo común divisor
divisor_comun = gcd(numerador, denominador)
self.numerador = numerador // divisor_comun
self.denominador = denominador // divisor_comun

def __str__(self):
return f'{self.numerador}/{self.denominador}'

def __add__(self, otra_fraccion):
# Suma de fracciones: (a/b) + (c/d) = (ad + bc) / bd
nuevo_numerador = (self.numerador * otra_fraccion.denominador) + \
(otra_fraccion.numerador * self.denominador)
nuevo_denominador = self.denominador * otra_fraccion.denominador
return Fraccion(nuevo_numerador, nuevo_denominador)

def __mul__(self, otra_fraccion):
# Multiplicación de fracciones: (a/b) * (c/d) = (ac) / (bd)
nuevo_numerador = self.numerador * otra_fraccion.numerador
nuevo_denominador = self.denominador * otra_fraccion.denominador
return Fraccion(nuevo_numerador, nuevo_denominador)

# Crear dos objetos Fraccion
frac1 = Fraccion(1, 2)
frac2 = Fraccion(3, 4)

# Sumar dos fracciones usando el operador +
suma = frac1 + frac2
print(f'Suma: {frac1} + {frac2} = {suma}')

# Multiplicar dos fracciones usando el operador *
producto = frac1 * frac2
print(f'Multiplicación: {frac1} * {frac2} = {producto}')
```

En este ejemplo, se definió la clase Fraccion para representar números fraccionarios y se sobrecargaron los operadores + y * para que realicen la suma y multiplicación de fracciones según las reglas matemáticas correspondientes. La simplificación de fracciones se realiza en el constructor utilizando el máximo común divisor.

Con estos métodos, al crear dos objetos Fraccion (frac1 y frac2) se pueden usar los operadores + y * para realizar operaciones aritméticas con fracciones de manera más clara e intuitiva que invocando métodos con nombre.

Capítulo 9

Composición

9.1. Introducción

La composición es un principio fundamental que se utiliza para crear objetos complejos al combinar o componer a partir de objetos más simples. La composición es un tipo de relación entre clases en la que un objeto se compone de otros objetos como sus partes constituyentes.

La composición se basa en la idea de que un objeto puede tener referencias a otros objetos como sus atributos, y puede utilizar sus métodos para realizar acciones específicas.

9.2. Cardinalidad

Dependiendo de la cantidad de objetos involucrados en una relación de composición, la implementación puede variar sensiblemente. Si la cardinalidad es 0, 1 o alguna cantidad constante, puede implementarse mediante el uso de referencias simples, de forma tal que en la clase compuesta existan atributos cuyo tipo sea el de la clase contenida. Por ejemplo, en una clase que represente un partido de algún deporte de equipos, puede plantearse que un objeto de la clase Partido, estará compuesto de dos (y únicamente dos) instancias de la clase Equipo.

Cuando la cardinalidad indica que el objeto compuesto se relaciona con una cantidad grande, variable o desconocida de objetos de la clase relacionada, la implementación se orienta a disponer de una colección de objetos. Siguiendo con el ejemplo anterior, un equipo estará compuesto de muchos jugadores, por lo tanto la clase Equipo puede tener un atributo de tipo lista o diccionario que almacene instancias de la clase Jugador.

9.3. Implementación

Para implementar composición en Python se puede plantear un esquema similar al siguiente:

- En la clase Contenedora agregar un atributo de alguno de los tipos de colección.
- En el constructor de la clase contenedora no recibir un parámetro para asignar la colección, únicamente debe ser instanciada vacía.
- No agregar métodos de asignación y consulta para la colección.
- Agregar un método de agregado a la colección, que reciba como parámetro una instancia de la clase Contenida.
- Todas las responsabilidades que involucren a la colección, deben ser programadas como métodos en la clase Contenedora.

9.4. Ejemplo

Suponiendo una relación de composición entre las clases Biblioteca y Libro, estableciendo que una biblioteca está compuesta de 0 o más libros, se puede plantear la siguiente implementación:

```
class Libro:
def __init__(self, titulo, autor):
self.titulo = titulo
self.autor = autor

class Biblioteca:
def __init__(self):
```

```
    self.libros = []

    def agregar_libro(self, libro):
    self.libros.append(libro)

    def mostrar_libros(self):
    for libro in self.libros:
    print(f"Título: {libro.titulo}, Autor: {libro.autor}")

    # Crear libros
    libro1 = Libro("El Gran Gatsby", "F. Scott Fitzgerald")
    libro2 = Libro("Cien Años de Soledad", "Gabriel García Márquez")

    # Crear una biblioteca y agregar libros
    biblioteca = Biblioteca()
    biblioteca.agregar_libro(libro1)
    biblioteca.agregar_libro(libro2)

    # Mostrar los libros en la biblioteca
    biblioteca.mostrar_libros()
```

Capítulo 10

Herencia

10.1. Introducción

La herencia es un concepto fundamental en la programación orientada a objetos que establece una relación esencial entre clases. Este mecanismo se aplica cuando existen clases que, aunque comparten similitudes notables, no son lo suficientemente idénticas como para ser consideradas como una única clase. En esta situación, la herencia entra en juego para vincular estas clases de manera que los objetos pertenecientes a ambas puedan ser utilizados de manera equivalente, como si efectivamente fueran instancias de la misma clase.

El proceso de herencia posibilita la creación de una nueva clase basada en una clase preexistente, sin la necesidad de duplicar y repetir el código de la clase original. De esta manera, se evita la tarea de copiar y pegar la implementación de una clase para crear una nueva, lo que redundaría en un mantenimiento engorroso y potencialmente propenso a errores.

La herencia implica que una clase derivada (o subclase) hereda propiedades y métodos de una clase base (o superclase). La subclase puede agregar nuevas funcionalidades o modificar las existentes, pero también hereda la estructura y el comportamiento de la superclase.

Esto es útil porque permite organizar y abstraer el código de manera eficiente. Por ejemplo, si se está desarrollando un programa para modelar vehículos, se podría tener una clase base llamada “Vehículo” con propiedades y métodos genéricos como “acelerar” y “frenar”. Luego, se pueden crear subclases tales como “Automóvil” y “Motocicleta” que heredan estas características básicas y agregan sus propias particularidades, ya sea agregando nuevos atributos y métodos o modificando métodos heredados brindando una implementación específica para sus responsabilidades y necesidades.

Al utilizar la herencia, se puede considerar a los objetos pertenecientes a ambas clases como si fueran objetos de la clase base. Esto implica que los objetos heredados heredan tanto las propiedades como los métodos de la clase base, lo que simplifica la organización del código y fomenta la reutilización eficiente de funcionalidades comunes.

10.2. “Es un”

El concepto de la herencia en la programación orientada a objetos se manifiesta mediante la capacidad de definir una clase a partir de otra, estableciendo una relación específica entre ellas. En este contexto, una de las clases involucradas se denomina **clase base**, mientras que la otra se conoce como **clase derivada**.

La clase derivada puede conceptualizarse como un caso especial o una variante de la clase base, lo que implica que todos los objetos pertenecientes a la clase derivada pueden ser considerados como objetos de la clase base. Esta relación se expresa de manera precisa con la afirmación: “un objeto de la clase derivada ES UN objeto de la clase base”. En general, es una práctica útil verificar las relaciones de herencia con esa frase: si la frase no tiene sentido, generalmente es síntoma de que las clases no deben estar relacionadas o lo hacen pero por otro tipo de relación.

Por ejemplo:

Clase base Cliente y Clase derivada ClienteOnline En este contexto, afirmar que “un cliente online ES UN cliente” es totalmente válido. La clase derivada ClienteOnline representa una variante específica de la clase base Cliente, pero todos los clientes en línea son, en última instancia, clientes en sí mismos.

Clase base Auto y Clase derivada Rueda La afirmación “una rueda ES UN auto” es inevitablemente incorrecta, como también lo es “un auto ES UNA rueda”. Esta situación pone en evidencia que tales clases no pueden estar relacionadas mediante herencia. Sin embargo, pueden estar relacionadas mediante algún otro tipo de relación, por caso, composición.

10.3. Implementación

10.3.1. Bloque class

Para implementar herencia en Python no se agregan muchos elementos sintácticos, en general las clases se redactan de una forma bastante similar a aquellas no involucradas en este tipo de relación.

En la clase base no se debe realizar ninguna modificación si se identifica la necesidad de una clase derivada. En la clase derivada se indica en la cabecera del bloque class, a continuación del nombre de la misma, el nombre de la clase base encerrado entre paréntesis:

```
class Derivada(Base):
```

10.3.2. Función super

En los métodos de las clases derivadas se dispone de una referencia denominada `super` la cual referencia al objeto que recibió el mensaje, pero permitiendo acceder únicamente a los miembros heredados. Es, por lo tanto, muy similar a self, pero no requiere ser recibida como un parámetro.

La referencia super es ocasionalmente utilizada de forma errónea para acceder a cualquier miembro declarado en la clase base. Esto evidencia una notoria ignorancia del concepto de la herencia: todo miembro de la clase base es heredado y por lo tanto accesible mediante la referencia self.

El uso de super debe estar focalizado en acceder a aquellos miembros que no pueden ser alcanzados mediante self. Esta situación ocurre puntualmente con dos casos bien delimitados. El método constructor __init__ no es heredado y muy habitualmente es necesario invocar explícitamente desde la derivada al constructor de la base. Asimismo es necesario utilizar super para realizar llamadas a un método de la clase base desde su redefinición en la derivada, ya que si se lo intenta invocar con self, se iniciaría una llamada recursiva.

10.3.3. Constructor

El método constructor de cada clase generalmente recibe parámetros para asignar valores iniciales a los atributos. Cuando se programa un constructor en una clase derivada, el caso más usual es el de recibir parámetros para inicializar los atributos de la propia clase y también los heredados.

A continuación el constructor de la derivada puede realizar una invocación explícita al constructor de la clase base a través de la referencia super:

```
class Derivada(Base):
    def __init__(self, params):
        super.__init__(params)
```

Capítulo 11

Polimorfismo

11.1. Introducción

11.1.1. Concepto

El polimorfismo es un principio fundamental de la programación orientada a objetos que permite que objetos de diferentes clases sean tratados como objetos de una clase común. Esta característica facilita que un mismo método pueda comportarse de manera diferente según el tipo específico del objeto que lo recibe, proporcionando flexibilidad y extensibilidad al código.

El término ”polimorfismo”deriva del griego y significa ”muchas formas”, lo cual describe precisamente su naturaleza: un mismo nombre de método puede tener múltiples implementaciones según la clase que lo contenga. En el contexto de la herencia, el polimorfismo se manifiesta cuando las clases derivadas redefinen métodos heredados de la clase base, proporcionando implementaciones específicas que se ajusten a sus particulares necesidades y características.

11.1.2. Sobreescritura de métodos

La sobreescritura de métodos es el mecanismo mediante el cual una clase derivada proporciona una implementación específica de un método que ya existe en su clase base. En Python, la sintaxis para sobreescribir un método es extremadamente sencilla: simplemente se define un método en la clase derivada con el mismo nombre que el método de la clase base que se desea redefinir. No se requieren palabras clave especiales ni anotaciones adicionales.

```
class Vehiculo:
    def acelerar(self):
        print("El vehículo está acelerando")

    def describir(self):
        print("Este es un vehículo genérico")

class Automovil(Vehiculo):
    def acelerar(self):
        print("El automóvil acelera suavemente")

    def describir(self):
        print("Este es un automóvil con cuatro ruedas")

class Motocicleta(Vehiculo):
    def acelerar(self):
        print("La motocicleta acelera rápidamente")
```

11.1.3. Uso de super

El método especial `__str__` es particularmente útil para demostrar el uso de `super()` en el contexto del polimorfismo, ya que frecuentemente es necesario extender la representación de cadena de la clase base en lugar de reemplazarla completamente.

Cuando se sobreescribe el método `__str__` en una clase derivada, es común utilizar `super()` para invocar la implementación de la clase base y luego agregar información específica de la clase derivada. Esto permite construir representaciones de cadena más completas y jerárquicas.

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

    def __str__(self):
        return f"Persona: {self.nombre}, {self.edad} años"

class Estudiante(Persona):
    def __init__(self, nombre, edad, carrera):
        super().__init__(nombre, edad)
        self.carrera = carrera

    def __str__(self):
        return f"{super().__str__()}, Carrera: {self.carrera}"

class Profesor(Persona):
    def __init__(self, nombre, edad, materia):
        super().__init__(nombre, edad)
        self.materia = materia

    def __str__(self):
        return f"{super().__str__()}, Materia: {self.materia}"
```

En este ejemplo, las clases `Estudiante` y `Profesor` extienden la representación de cadena de la clase `Persona` agregando información específica. El uso de `super().__str__()` permite reutilizar la lógica de la clase base y evitar la duplicación de código.

11.2. Clases y métodos abstractos

11.2.1. Concepto

Una clase abstracta es aquella que no puede ser instanciada directamente y que generalmente contiene uno o más métodos abstractos. Los métodos abstractos son declaraciones de métodos que deben ser implementados obligatoriamente por las clases derivadas, pero que no poseen implementación en la clase base.

Las clases abstractas sirven como plantillas o contratos que definen qué métodos deben implementar las clases derivadas, garantizando así una interfaz común entre diferentes implementaciones. Esto es especialmente útil cuando se desea definir un comportamiento común pero la implementación específica varía según cada subclase.

11.2.2. Uso del Decorador @abstractmethod

Python proporciona el módulo `abc` (Abstract Base Classes) para implementar clases y métodos abstractos. Para crear un método abstracto, se utiliza el decorador `@abstractmethod` y la clase debe heredar de `ABC`.

```python
from abc import ABC, abstractmethod

class Figura(ABC):
    def __init__(self, nombre):
        self.nombre = nombre

    @abstractmethod
    def calcular_area(self):
        pass

    @abstractmethod
    def calcular_perimetro(self):
        pass

    def describir(self):
        return f"Figura: {self.nombre}"

class Rectangulo(Figura):
    def __init__(self, ancho, alto):
        super().__init__("Rectángulo")
        self.ancho = ancho
        self.alto = alto

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

```
    def calcular_perimetro(self):
        return 2 * (self.ancho + self.alto)

class Circulo(Figura):
    def __init__(self, radio):
        super().__init__("Círculo")
        self.radio = radio

    def calcular_area(self):
        return 3.14159 * self.radio ** 2

    def calcular_perimetro(self):
        return 2 * 3.14159 * self.radio
```

Si se intenta instanciar la clase `Figura` directamente o crear una subclase que no implemente todos los métodos abstractos, Python generará un error `TypeError`. Esto garantiza que todas las clases derivadas cumplan con el contrato establecido por la clase abstracta.

11.3. Interfaces

11.3.1. Implementación

Las interfaces definen un conjunto de métodos que una clase debe implementar, sin proporcionar la implementación de dichos métodos. En UML, las interfaces se representan como contratos que especifican qué servicios debe proporcionar una clase, pero no cómo debe proporcionarlos.

Aunque Python no tiene una palabra clave específica para interfaces como otros lenguajes, se pueden implementar utilizando clases abstractas que contengan únicamente métodos abstractos. Esta aproximación permite replicar el concepto de interfaces de UML en Python.

```
from abc import ABC, abstractmethod

class IReproductorMultimedia(ABC):
    @abstractmethod
    def reproducir(self):
        pass

    @abstractmethod
    def pausar(self):
        pass

    @abstractmethod
    def detener(self):
        pass

class IAlmacenamiento(ABC):
    @abstractmethod
    def guardar(self, datos):
        pass

    @abstractmethod
    def cargar(self):
        pass

class ReproductorMP3(IReproductorMultimedia, IAlmacenamiento):
    def __init__(self):
        self.estado = "detenido"
        self.archivos = []

    def reproducir(self):
        self.estado = "reproduciendo"
        print("Reproduciendo archivo MP3")

    def pausar(self):
        self.estado = "pausado"
        print("Reproducción pausada")

    def detener(self):
        self.estado = "detenido"
        print("Reproducción detenida")
```

```
    def guardar(self, archivo):
        self.archivos.append(archivo)
        print(f"Archivo {archivo} guardado")

    def cargar(self):
        return self.archivos
```

11.3.2. Implementación de múltiples interfaces

Python permite que una clase implemente múltiples interfaces mediante herencia múltiple, lo cual es equivalente al concepto de implementación múltiple de interfaces en UML. Esto proporciona gran flexibilidad para diseñar clases que cumplan con varios contratos diferentes.

```
class IVehiculoTerrestre(ABC):
    @abstractmethod
    def conducir(self):
        pass

class IVehiculoAcuatico(ABC):
    @abstractmethod
    def navegar(self):
        pass

class VehiculoAnfibio(IVehiculoTerrestre, IVehiculoAcuatico):
    def __init__(self, modelo):
        self.modelo = modelo

    def conducir(self):
        print(f"{self.modelo} está conduciendo en tierra")

    def navegar(self):
        print(f"{self.modelo} está navegando en agua")
```

Esta implementación permite que un objeto `VehiculoAnfibio` sea tratado polimórficamente como un `IVehiculoTerrestre` o como un `IVehiculoAcuatico`, dependiendo del contexto en el que se utilice, replicando así el comportamiento de las interfaces múltiples de UML en Python.

Capítulo 12

Testing unitario

12.1. Introducción

El testing unitario es una práctica fundamental en el desarrollo de software que consiste en escribir y ejecutar pruebas automatizadas para verificar que las unidades individuales de código funcionen correctamente de manera aislada. Una unidad de código típicamente corresponde a un método, función o clase, y representa la porción más pequeña de código que puede ser probada de forma independiente.

El propósito principal del testing unitario es detectar errores en las primeras etapas del desarrollo, cuando son más fáciles y económicos de corregir. Al verificar que cada componente individual funciona según lo esperado, se puede tener mayor confianza en que el sistema completo funcionará correctamente cuando todas las partes se integren.

En la programación orientada a objetos, el testing unitario cobra especial relevancia debido a la naturaleza modular y encapsulada de las clases. Cada clase debe cumplir con un conjunto específico de responsabilidades y mantener su estado interno de manera coherente. Las pruebas unitarias permiten verificar que los métodos de una clase produzcan los resultados esperados, que las validaciones internas funcionen correctamente, y que el encapsulamiento se mantenga adecuadamente.

Los beneficios del testing unitario incluyen la detección temprana de errores, la facilitación del refactoring seguro del código, la documentación implícita del comportamiento esperado de cada componente, y la reducción del tiempo necesario para localizar y corregir defectos. Además, las pruebas unitarias actúan como una red de seguridad que permite a los desarrolladores modificar el código con confianza, sabiendo que cualquier regresión será detectada inmediatamente.

Para que el testing unitario sea efectivo, las pruebas deben ser rápidas de ejecutar, independientes entre sí, repetibles y fáciles de mantener. Cada prueba debe verificar un comportamiento específico y proporcionar información clara sobre qué aspecto del código falló cuando ocurre un error.

12.2. Pytest

Pytest es un framework de testing para Python que se ha convertido en el estándar de facto para escribir y ejecutar pruebas unitarias en este lenguaje. A diferencia de otros frameworks de testing, pytest se destaca por su sintaxis simple y natural, su capacidad de autodescubrimiento de pruebas, y sus potentes características avanzadas que facilitan la escritura de pruebas comprehensivas.

Una de las principales ventajas de pytest es que permite escribir pruebas utilizando la sintaxis estándar de Python con la instrucción `assert`, eliminando la necesidad de aprender métodos específicos del framework como ocurre con otros sistemas de testing. Esto hace que las pruebas sean más legibles y fáciles de escribir para desarrolladores que ya conocen Python.

Para instalar pytest, se utiliza el gestor de paquetes pip mediante el siguiente comando:

```
pip install pytest
```

Una vez instalado, pytest puede ejecutarse desde la línea de comandos. El framework automáticamente descubre y ejecuta todas las pruebas en el directorio actual y sus subdirectorios. Para ejecutar las pruebas, simplemente se utiliza:

```
pytest
```

Pytest busca automáticamente archivos que comiencen con `test_` o terminen con `_test.py`, y dentro de estos archivos ejecuta todas las funciones que comiencen con `test_`. Esta convención de nomenclatura facilita la organización y el descubrimiento automático de las pruebas.

Además del comando básico, pytest ofrece múltiples opciones de ejecución. Por ejemplo, se puede ejecutar un archivo específico de pruebas, mostrar información más detallada sobre la ejecución, o ejecutar solo las pruebas que contienen una palabra clave específica:

```
# Ejecutar un archivo específico
pytest test_ejemplo.py

# Mostrar información detallada
pytest -v

# Ejecutar pruebas que contengan una palabra clave
pytest -k "suma"
```

Pytest también genera reportes claros y detallados cuando las pruebas fallan, mostrando exactamente qué assertion falló, los valores esperados versus los obtenidos, y el contexto completo del error.

12.3. Pruebas

La estructura fundamental de las pruebas en pytest se basa en funciones que contienen assertions para verificar el comportamiento esperado del código. Una suite de pruebas es simplemente un conjunto de funciones de prueba organizadas en uno o más archivos, mientras que cada caso de prueba individual corresponde a una función específica que verifica un aspecto particular del código bajo prueba.

La sintaxis básica para escribir una prueba en pytest es crear una función cuyo nombre comience con `test_` y utilizar la instrucción **`assert`** para verificar condiciones:

```
def test_ejemplo_basico():
    resultado = 2 + 2
    assert resultado == 4
```

Para organizar las pruebas relacionadas con una clase específica, se recomienda crear un archivo de pruebas separado, típicamente con el nombre `test_nombre_clase.py`. Dentro de este archivo, se pueden agrupar las pruebas utilizando clases de prueba, aunque esto no es obligatorio:

```
class TestCalculadora:
    def test_suma(self):
        calc = Calculadora()
        resultado = calc.sumar(2, 3)
        assert resultado == 5

    def test_division_por_cero(self):
        calc = Calculadora()
        with pytest.raises(ZeroDivisionError):
            calc.dividir(10, 0)
```

12.3.1. Aserciones de igualdad

Las aserciones de igualdad son las más comunes en las pruebas unitarias y verifican que un valor obtenido sea exactamente igual al valor esperado. Pytest utiliza la instrucción **`assert`** estándar de Python junto con el operador de igualdad:

```
def test_igualdad_numerica():
    assert 5 == 5
    assert 2.5 == 2.5
```

```
def test_igualdad_cadenas():
    assert "hola" == "hola"
    assert "Python".lower() == "python"

def test_igualdad_listas():
    assert [1, 2, 3] == [1, 2, 3]
    assert sorted([3, 1, 2]) == [1, 2, 3]
```

Para la desigualdad, se utiliza el operador $\neq$:

```
def test_desigualdad():
    assert 5 ≠ 3
    assert "hola" ≠ "adiós"
```

12.3.2. Aserciones de comparación

Las aserciones de comparación permiten verificar relaciones de orden entre valores utilizando los operadores de comparación estándar de Python:

```
def test_comparaciones_numericas():
    assert 5 > 3
    assert 2 < 10
    assert 5 ≥ 5
    assert 3 ≤ 7

def test_comparaciones_cadenas():
    assert "abc" < "def"  # Orden alfabético
    assert "z" > "a"
```

12.3.3. Aserciones de pertenencia

Las aserciones de pertenencia verifican si un elemento está contenido en una colección utilizando el operador `in`:

```
def test_pertenencia_listas():
    numeros = [1, 2, 3, 4, 5]
    assert 3 in numeros
    assert 6 not in numeros

def test_pertenencia_cadenas():
    texto = "Hola mundo"
    assert "mundo" in texto
    assert "Python" not in texto

def test_pertenencia_diccionarios():
    datos = {"nombre": "Juan", "edad": 25}
    assert "nombre" in datos
    assert "apellido" not in datos
```

12.3.4. Aserciones de tipo

Las aserciones de tipo verifican que un objeto sea de un tipo específico utilizando la función `isinstance()`:

```
def test_tipos():
    assert isinstance(5, int)
    assert isinstance(3.14, float)
    assert isinstance("texto", str)
    assert isinstance([1, 2, 3], list)
    assert isinstance({"a": 1}, dict)
```

12.3.5. Aserciones de excepciones

Para verificar que el código lance excepciones bajo ciertas condiciones, pytest proporciona el context manager `pytest.raises()`:

```python
import pytest

def test_excepcion_division_cero():
    with pytest.raises(ZeroDivisionError):
        resultado = 10 / 0

def test_excepcion_valor_incorrecto():
    with pytest.raises(ValueError):
        int("texto_invalido")

def test_excepcion_con_mensaje():
    with pytest.raises(ValueError, match="invalid literal"):
        int("abc")
```

12.3.6. Aserciones de aproximación

Para valores de punto flotante, donde la comparación exacta puede fallar debido a imprecisiones de representación, pytest ofrece `pytest.approx()`:

```python
import pytest

def test_aproximacion():
    assert 0.1 + 0.2 == pytest.approx(0.3)
    assert 3.14159 == pytest.approx(3.14, rel=1e-2)
```

12.4. Ejemplo

Para ilustrar la aplicación práctica del testing unitario con pytest, se presenta una clase `Numeros` que encapsula una colección de números y proporciona métodos para realizar operaciones estadísticas básicas. Esta clase implementa el principio de composición al contener una lista de números como atributo interno.

```python
class Numeros:
    def __init__(self):
        self._numeros = []

    def agregar(self, numero):
        if not isinstance(numero, (int, float)):
            raise TypeError("Solo se pueden agregar números")
        self._numeros.append(numero)

    def cantidad(self):
        return len(self._numeros)

    def suma(self):
        return sum(self._numeros)

    def promedio(self):
        if self.cantidad() == 0:
            raise ValueError("No se puede calcular el promedio de una colección vacía")
        return self.suma() / self.cantidad()

    def maximo(self):
        if self.cantidad() == 0:
            raise ValueError("No se puede obtener el máximo de una colección vacía")
        return max(self._numeros)

    def minimo(self):
        if self.cantidad() == 0:
            raise ValueError("No se puede obtener el mínimo de una colección vacía")
        return min(self._numeros)

    def contiene(self, numero):
        return numero in self._numeros

    def limpiar(self):
        self._numeros.clear()
```

A continuación se presenta una suite completa de pruebas para esta clase, organizada de manera sistemática para cubrir todos los casos de uso posibles, incluyendo casos límite y situaciones de error:

```python
import pytest
from numeros import Numeros

class TestNumeros:

    def test_coleccion_vacia_inicial(self):
        """Verifica que una nueva instancia esté vacía"""
        coleccion = Numeros()
        assert coleccion.cantidad() == 0

    def test_agregar_numero_entero(self):
        """Verifica que se puedan agregar números enteros"""
        coleccion = Numeros()
        coleccion.agregar(5)
        assert coleccion.cantidad() == 1
        assert coleccion.contiene(5)

    def test_agregar_numero_flotante(self):
        """Verifica que se puedan agregar números flotantes"""
        coleccion = Numeros()
        coleccion.agregar(3.14)
        assert coleccion.cantidad() == 1
        assert coleccion.contiene(3.14)

    def test_agregar_numero_negativo(self):
        """Verifica que se puedan agregar números negativos"""
        coleccion = Numeros()
        coleccion.agregar(-10)
        assert coleccion.cantidad() == 1
        assert coleccion.contiene(-10)

    def test_agregar_cero(self):
        """Verifica que se pueda agregar el número cero"""
        coleccion = Numeros()
        coleccion.agregar(0)
        assert coleccion.cantidad() == 1
        assert coleccion.contiene(0)

    def test_agregar_tipo_invalido_cadena(self):
        """Verifica que se lance excepción al agregar una cadena"""
        coleccion = Numeros()
        with pytest.raises(TypeError, match="Solo se pueden agregar números"):
            coleccion.agregar("cinco")

    def test_agregar_tipo_invalido_lista(self):
        """Verifica que se lance excepción al agregar una lista"""
        coleccion = Numeros()
        with pytest.raises(TypeError):
            coleccion.agregar([1, 2, 3])

    def test_agregar_multiple_numeros(self):
        """Verifica que se puedan agregar múltiples números"""
        coleccion = Numeros()
        numeros = [1, 2, 3, 4, 5]
        for numero in numeros:
            coleccion.agregar(numero)

        assert coleccion.cantidad() == 5
        for numero in numeros:
            assert coleccion.contiene(numero)

    def test_suma_un_numero(self):
        """Verifica la suma con un solo número"""
        coleccion = Numeros()
        coleccion.agregar(7)
        assert coleccion.suma() == 7

    def test_suma_multiples_numeros(self):
        """Verifica la suma con múltiples números"""
        coleccion = Numeros()
        coleccion.agregar(2)
        coleccion.agregar(3)
        coleccion.agregar(5)
        assert coleccion.suma() == 10

    def test_suma_numeros_negativos(self):
        """Verifica la suma con números negativos"""
        coleccion = Numeros()
```

```
        coleccion.agregar(-2)
        coleccion.agregar(5)
        coleccion.agregar(-1)
        assert coleccion.suma() == 2

    def test_suma_coleccion_vacia(self):
        """Verifica que la suma de una colección vacía sea cero"""
        coleccion = Numeros()
        assert coleccion.suma() == 0

    def test_promedio_un_numero(self):
        """Verifica el promedio con un solo número"""
        coleccion = Numeros()
        coleccion.agregar(8)
        assert coleccion.promedio() == 8

    def test_promedio_multiples_numeros(self):
        """Verifica el promedio con múltiples números"""
        coleccion = Numeros()
        coleccion.agregar(2)
        coleccion.agregar(4)
        coleccion.agregar(6)
        assert coleccion.promedio() == 4

    def test_promedio_con_flotantes(self):
        """Verifica el promedio con números flotantes"""
        coleccion = Numeros()
        coleccion.agregar(1.5)
        coleccion.agregar(2.5)
        assert coleccion.promedio() == pytest.approx(2.0)

    def test_promedio_coleccion_vacia(self):
        """Verifica que se lance excepción al calcular promedio de colección vacía"""
        coleccion = Numeros()
        with pytest.raises(ValueError, match="No se puede calcular el promedio de una colección vacía"):
            coleccion.promedio()

    def test_maximo_un_numero(self):
        """Verifica el máximo con un solo número"""
        coleccion = Numeros()
        coleccion.agregar(42)
        assert coleccion.maximo() == 42

    def test_maximo_multiples_numeros(self):
        """Verifica el máximo con múltiples números"""
        coleccion = Numeros()
        numeros = [3, 1, 4, 1, 5, 9, 2, 6]
        for numero in numeros:
            coleccion.agregar(numero)
        assert coleccion.maximo() == 9

    def test_maximo_numeros_negativos(self):
        """Verifica el máximo con solo números negativos"""
        coleccion = Numeros()
        coleccion.agregar(-5)
        coleccion.agregar(-2)
        coleccion.agregar(-8)
        assert coleccion.maximo() == -2

    def test_maximo_coleccion_vacia(self):
        """Verifica que se lance excepción al buscar máximo de colección vacía"""
        coleccion = Numeros()
        with pytest.raises(ValueError, match="No se puede obtener el máximo de una colección vacía"):
            coleccion.maximo()

    def test_minimo_un_numero(self):
        """Verifica el mínimo con un solo número"""
        coleccion = Numeros()
        coleccion.agregar(17)
        assert coleccion.minimo() == 17

    def test_minimo_multiples_numeros(self):
        """Verifica el mínimo con múltiples números"""
        coleccion = Numeros()
        numeros = [3, 1, 4, 1, 5, 9, 2, 6]
        for numero in numeros:
            coleccion.agregar(numero)
        assert coleccion.minimo() == 1
```

```
    def test_minimo_numeros_positivos(self):
        """Verifica el mínimo con solo números positivos"""
        coleccion = Numeros()
        coleccion.agregar(5)
        coleccion.agregar(2)
        coleccion.agregar(8)
        assert coleccion.minimo() == 2

    def test_minimo_coleccion_vacia(self):
        """Verifica que se lance excepción al buscar mínimo de colección vacía"""
        coleccion = Numeros()
        with pytest.raises(ValueError, match="No se puede obtener el mínimo de una colección vacía"):
            coleccion.minimo()

    def test_contiene_numero_presente(self):
        """Verifica que contiene devuelva True para números presentes"""
        coleccion = Numeros()
        coleccion.agregar(10)
        coleccion.agregar(20)
        coleccion.agregar(30)
        assert coleccion.contiene(20)

    def test_contiene_numero_ausente(self):
        """Verifica que contiene devuelva False para números ausentes"""
        coleccion = Numeros()
        coleccion.agregar(10)
        coleccion.agregar(20)
        assert not coleccion.contiene(15)

    def test_contiene_coleccion_vacia(self):
        """Verifica que contiene devuelva False en colección vacía"""
        coleccion = Numeros()
        assert not coleccion.contiene(5)

    def test_limpiar_coleccion(self):
        """Verifica que limpiar vacíe completamente la colección"""
        coleccion = Numeros()
        coleccion.agregar(1)
        coleccion.agregar(2)
        coleccion.agregar(3)

        assert coleccion.cantidad() == 3

        coleccion.limpiar()
        assert coleccion.cantidad() == 0
        assert not coleccion.contiene(1)
        assert not coleccion.contiene(2)
        assert not coleccion.contiene(3)

    def test_operaciones_despues_limpiar(self):
        """Verifica que las operaciones fallen después de limpiar"""
        coleccion = Numeros()
        coleccion.agregar(5)
        coleccion.limpiar()

        with pytest.raises(ValueError):
            coleccion.promedio()

        with pytest.raises(ValueError):
            coleccion.maximo()

        with pytest.raises(ValueError):
            coleccion.minimo()

    def test_flujo_completo(self):
        """Prueba de integración que verifica un flujo completo de operaciones"""
        coleccion = Numeros()

        # Agregar números
        numeros = [2, 4, 6, 8, 10]
        for numero in numeros:
            coleccion.agregar(numero)

        # Verificar todas las operaciones
        assert coleccion.cantidad() == 5
        assert coleccion.suma() == 30
        assert coleccion.promedio() == 6
```

```
        assert coleccion.maximo() == 10
        assert coleccion.minimo() == 2

        # Verificar pertenencia
        for numero in numeros:
            assert coleccion.contiene(numero)

        assert not coleccion.contiene(7)
```

Esta suite de pruebas cubre exhaustivamente todos los métodos de la clase `Numeros`, incluyendo casos normales, casos límite y situaciones de error. Las pruebas están organizadas de manera lógica, cada una verifica un comportamiento específico, y utilizan nombres descriptivos que facilitan la comprensión de qué está siendo probado.

El conjunto de pruebas incluye verificaciones para el manejo correcto de excepciones, validaciones de tipos, operaciones con colecciones vacías, y un caso de prueba de integración que verifica el flujo completo de operaciones. Esta aproximación sistemática al testing garantiza que la clase `Numeros` funcione correctamente en todas las situaciones previstas.