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

```
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 requerirían 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 modi-

ficar, 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()
```