“Año del Bicentenario, de la consolidación de nuestra Independencia, y de la conmemoración de las heroicas batallas de Junín y Ayacucho”

Universidad Nacional Mayor de San Marcos

Universidad del Perú, Decana de América

Facultad de Ingeniería de Sistemas e Informática

Escuela Profesional de Ingeniería de Software

Introducción a la programación con Python

Kevin Tupac Agüero

Índice

1. Python

1.1. ¿Que es Python?

Python es un lenguaje de programación de alto nivel, interpretado y de propósito general. Fue creado por Guido van Rossum y lanzado por primera vez en 1991. Python se destaca por su sintaxis clara y legible, lo que facilita tanto su aprendizaje como su uso.

1.2. Características de Python

- Sintaxis: Fácil de leer y escribir, amigable con el programador.
- Interpretado: Se ejecuta de manera procedimental, facilitando la prueba y depuración.
- Multiplataforma: Python puede ejecutarse en diferentes sistemas operativos.
- Propósito General: Utilizado en diferentes entornos de trabajo, desde scripts hasta Desarrollo Web, Machine Learning e IA.
- Herramientas de Desarrollo: Gran cantidad de bibliotecas[1], paquetes[2] y módulos[3], además cuenta con muchos Frameworks[4] que facilitan el desarrollo de software.
- Comunidad Activa: Python cuenta con una comunidad de desarrolladores, los cuales contribuyen con la mejora del lenguaje, al este ser de código libre. Por tal razón está en continuas mejoras y corrección de errores, como también creación de nuevas bibliotecas y Frameworks.

1.3. Aplicaciones de Python

- Desarrollo Web: Cuenta con Frameworks para la creación de páginas y servidores web, dándole competencia a otros lenguajes más especializados para este campo.
- Ciencia de Datos y Análisis de Datos: Cuenta con herramientas muy poderosas para la implementación de probabilidades y estadística, contando también con la capacidad de generar todo tipo de gráficos, además el manejo de gran cantidad de datos, todo gracias a sus bibliotecas.
- IA y ML: La capacidad de crear algo que otros lenguajes no pueden, Python se convierte en pilar para la creación de Inteligencia Artificial y desarrollo de Machine Learning, con la capacidad de cómputo para manejar grandes cantidades de cálculos matemáticos, ya sean multiplicación de matrices de n dimensiones.
- Automatización: Con la capacidad de automatizar funciones dentro de los servidores, para ahorrar el trabajo repetitivo y mejorando la eficiencia.
- Computación Científica: Permite la realización de simulaciones de la vida real, gracias a su poder de procesamiento de cálculos grandes y dinámicos.

[1] Conjuntos de funciones y recursos reutilizables.

[2] Grupos de módulos organizados en carpetas.

[3] Archivos individuales de código que contienen funciones y clases.

[4] Conjuntos de herramientas y reglas para desarrollar aplicaciones estructuradas.

- Ciberseguridad: Utilizado para crear las herramientas para la penetración y análisis de seguridad.

2. Instalación de Python

2.1. Descargar Python

Ingrese a la página Oficial de Python y descargue[5] la versión que esté con el término de seguridad, ya que se van agregando nuevas versiones, pero aún cuentan con fallos, se recomienda descargar 1 o 2 versiones anteriores a la actual, además seleccione el instalador dependiendo de la cantidad de bits que tiene su equipo y el sistema operativo en donde se encuentre.
Nota: Para la ejecución de todos los códigos que se muestran como ejemplos, se está utilizando la versión 3.11.9 de Python.

- **En Windows**:
  1. Ejecuta el instalador descargado.
  2. Marca la opción ‘‘Add Python to PATH’’.
  3. Haz clic en ‘‘Install Now’’.

2.2. Configurar el PATH

Configurar el PATH es importante para poder ejecutar Python desde cualquier terminal o línea de comandos. Aquí están las instrucciones para cada sistema operativo:

1. Abre el menú de Windows y busca ‘‘Editar las variables del entorno del sistema’’.
2. Haz clic en ‘‘Variables de entorno’’.
3. En la sección ‘‘Variables del sistema’’, busca la variable ‘‘Path’’ y selecciona *’Editar’*.
4. Haz clic en ‘‘Nuevo’’ y añade la ruta al directorio donde está instalado Python y el directorio de scripts. Por ejemplo:
   - C:\Users\kevin\AppData\Local\Programs\Python\Python311
   - C:\Users\kevin\AppData\Local\Programs\Python\Python311\Scripts
5. Haz clic en ‘‘Aceptar’’ en todas las ventanas para guardar los cambios.
6. Ingresa a la terminal mediante Ctrl + R e ingresa a cmd, ahi pon el comando:

```
#python3 --version
C:\Users\kevin>python3 --version
Python 3.11.9
```

Código 1: Verificación de versión de Python

Con estos pasos podrá verificar que ya tiene instalado Python en su computador.

[5] https://www.python.org/downloads/

2.3. Instalación en Linux

La instalación es mediante terminal de Linux, en este caso estamos dando por hecho que tiene una distribución de Ubuntu, pero si tiene otro distro, sería mediante otros comandos.

1. Actualiza los repositorios de paquetes:

```
sudo apt update
```

Código 2: Actualización de los paquetes

```
sudo apt install python3
```

Código 3: Instalación de Python

```
python3 --version
```

Código 4: Verificar versión Python en Linux

3. Entorno de trabajo

3.1. ¿Qué es un IDE?

Un Entorno de Desarrollo Integrado (IDE, por sus siglas en inglés) es una aplicación que proporciona un conjunto de herramientas para facilitar la creación, mantenimiento y prueba de software. Un IDE incluye un editor de código, herramientas de compilación, un depurador, y a menudo otras herramientas útiles como control de versiones, diseño de interfaces gráficas y más. Ejemplos de IDEs utilizados por la comunidad son Visual Studio, PyCharm, Eclipse e IntelliJ IDEA. El propósito principal de un IDE es proporcionar un entorno unificado y simplificado que permita a los desarrolladores escribir, probar y depurar su código, lo cual facilita el trabajo.

3.2. ¿Qué es depurar?

Depurar es el proceso de encontrar y resolver errores o problemas en el código de un programa. Estos errores pueden ser de diferentes tipos, como errores de sintaxis, errores de lógica, o errores de ejecución. La depuración implica el uso de herramientas y técnicas para identificar la causa de los problemas y corregirlos. Los IDEs suelen incluir depuradores que permiten a los desarrolladores ejecutar su código paso a paso, inspeccionar variables, y establecer puntos de interrupción para analizar el flujo de ejecución del programa.

3.3. ¿Qué es un compilador?

Es pasar nuestro código a ceros y unos para que sea entendible por la máquina. Un compilador es un programa que traduce el código fuente escrito en un lenguaje de programación de alto nivel (como C, C++ o Java) a un lenguaje de máquina o código binario que puede ser ejecutado directamente por la CPU de un ordenador. El proceso de compilación implica varias fases, incluyendo el análisis léxico, el análisis sintáctico, la optimización del código y la generación de código. El resultado es un archivo ejecutable que puede ser ejecutado en un sistema sin necesidad del código fuente original.

3.4. ¿Qué es un intérprete?

Un intérprete es un programa que ejecuta el código fuente de un programa directamente, línea por línea, sin necesidad de compilarlo previamente a código de máquina. Los intérpretes leen el código fuente, lo traducen a instrucciones ejecutables y ejecutan esas instrucciones en tiempo real. Ejemplos de lenguajes de programación que suelen utilizar intérpretes incluyen Python, JavaScript y Ruby. La principal ventaja de los intérpretes es que permiten una mayor flexibilidad y facilidad para probar y modificar el código rápidamente, aunque suelen ser menos eficientes en términos de velocidad de ejecución comparado con el código compilado.

4. Python Básico

4.1. Sintaxis

- En otros lenguajes de programación tienes que poner ; para terminar una línea de código, pero en Python no es necesario; en su lugar, Python utiliza la indentación para definir bloques de código. Esto permite escribir código de manera más clara y estructurada, sin preocuparnos por el punto y coma como en otros lenguajes.
- En Python, al momento de declarar una variable, se crea automáticamente el tipo de variable basado en el valor asignado, aunque también podemos asignarle el valor:
  - **int()**: Convierte un valor a entero.
  - **str()**: Convierte un valor a cadena.
  - **float()**: Convierte un valor a flotante.
  - **bool()**: Convierte un valor a booleano.
  - **list()**: Convierte un valor a lista.
  - **tuple()**: Convierte un valor a tupla.
  - **set()**: Convierte un valor a conjunto.
  - **dict()**: Convierte un valor a diccionario.

`Ejemplo:`

```
    # Declaracion automatica de tipos
    x = 5       # x es un entero
    y = "Hola"  # y es una cadena
    z = 3.14    # z es un flotante
    numero2 = ["2", 2, 2.0] # numero2 es una lista

    # Declaracion explicita de tipos
    a = int(5)               # a es un entero
    b = str("Hola")          # b es una cadena
    c = float(3.14)          # c es un flotante
    d = bool(1)              # d es un booleano
    e = list((1, 2, 3))      # e es una lista
    f = tuple([1, 2, 3])     # f es una tupla
    g = set([1, 2, 2, 3])    # g es un conjunto
    h = dict({1: "a", 2: "b"}) # h es un diccionario
```

Código 5: Utilización de asignación de tipo de variable

- Para comentar una línea de código se utiliza el símbolo #, pero para comentar una sección de código se utilizan tres comillas simples ' o tres comillas dobles ".

```
    # Esta es una linea comentada
    '''Esto es un comentario con comillas simples'''
    """Esto es un comentario con comillas dobles"""
```

Código 6: Ejemplo de comentarios en Python

- Existen palabras reservadas en Python. Estas palabras tienen un significado especial y no pueden ser utilizadas como nombres de variables. Algunas de estas palabras son:

`and`, `as`, `assert`, `break`, `class`, `continue`, `def`, `del`, `elif`, `else`, `except`, `False`, `finally`, `for`, `from`, `global`, `if`, `import`, `in`, `is`, `lambda`, `None`, `nonlocal`, `not`, `or`, `pass`, `raise`, `return`, `True`, `try`, `while`, `with`, `yield`

4.2. Tipos de Variables en Python

En Python, las variables pueden almacenar diferentes tipos de datos. A continuación, se describen los tipos de variables más comunes:

4.2.1. Enteros (int)

Los enteros son números sin punto decimal que están entre el rango desde -infinito a +infinito, siempre y cuando tu computadora tenga suficiente memoria para manejarlo.

```
    numero_entero = 42
    numero_grande = 10**100
    numero_negativo_grande = -10**100
    print(type(numero_entero))  # Salida: <class 'int'>
```

Código 7: Variable de tipo entero

4.2.2. Flotantes (float)

Los flotantes son números que tienen punto decimal. Según el estándar IEEE 754, el float está representado por una precisión doble(double precision), lo cual significa que tienen un rango y una precisión.

```
    numero_flotante = 3.14159
    flotante_pequeno = 2.2250738585072014e-308
    flotante_grande = 1.7976931348623157e+308
    preciso = 1.12345678901234567890 #Precision de un float

    print(type(numero_flotante))  # Salida: <class 'float'>

    # Pero si esta fuera del rango, nos dara error.

    # Numero demasiado grande
    demasiado_grande = 1.7976931348623157e+309
    print(demasiado_grande)  # Output: inf

    # Numero demasiado pequenio
    demasiado_pequeno = 2.2250738585072014e-309
    print(demasiado_pequeno)  # Output: 0.0
```

Código 8: Variable de tipo flotante

4.2.3. Cadenas (str)

Las cadenas son secuencias de caracteres encerradas entre comillas simples o dobles.

```
    cadena_simple = 'Hola, mundo!'
    cadena_doble = "Hola, mundo!"
    cadena_triple_simple = '''Hola, mundo!'''
    cadena_triple_doble = """Hola, mundo!"""
    letra = "a"
    print(type(letra))  # Salida: <class 'str'>
```

Código 9: Variable de tipo cadena

4.2.4. Booleanos (bool)

Los booleanos solo pueden tener dos valores: `True` o `False`.

```
    es_verdadero = True
    es_falso = False
    print(type(es_verdadero))  # Salida: <class 'bool'>
    print(type(es_falso))  # Salida: <class 'bool'>
```

Código 10: Variables de tipo booleano

4.2.5. Listas (list)

Las listas son colecciones ordenadas de elementos que pueden ser de diferentes tipos. Se definen utilizando corchetes `[ ]`.

```
    lista_numeros = [1, 2, 3, 4, 5]
    print(type(lista_numeros))  # Salida: <class 'list'>
```

Código 11: Variable de tipo lista

4.2.6. Tuplas (tuple)

Las tuplas son colecciones ordenadas de elementos que no se pueden modificar (inmutables). Se definen utilizando paréntesis `( )`.

```
    tupla_numeros = (1, 2, 3, 4, 5)
    print(type(tupla_numeros))  # Salida: <class 'tuple'>
```

Código 12: Variable de tipo tupla

4.2.7. Conjuntos (set)

Los conjuntos son colecciones desordenadas de elementos únicos. Se definen utilizando llaves {}.

```
    conjunto_numeros = {1, 2, 3, 4, 5}
    print(type(conjunto_numeros))  # Salida: <class 'set'>
```

Código 13: Variable de tipo conjunto

4.2.8. Diccionarios (dict)

Los diccionarios son colecciones desordenadas de pares clave-valor. Se definen utilizando llaves {}.

```
diccionario = {"nombre": "Juan", "edad": 30, "ciudad": "Lima"}
print(type(diccionario))  # Salida: <class 'dict'>
```

Código 14: Variable de tipo diccionario

4.3. Operadores Aritméticos

- + : Suma dos operandos.

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

Código 15: Operación de suma

- **-** : Resta el segundo operando del primero.

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

Código 16: Operación de resta

- * : Multiplica dos operandos.

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

Código 17: Operación de multiplicación

- // : Divide el primer operando por el segundo y devuelve la parte entera del resultado.

```
resultado = 15 // 2
print(resultado)  # Salida: 7
```

Código 18: Operación de división entera

- / : Divide el primer operando por el segundo.

```
resultado = 15 / 2
print(resultado)  # Salida: 7.5
```

Código 19: Operación de división

- ** : Eleva el primer operando a la potencia del segundo operando.

```
resultado = 2 ** 3
print(resultado)  # Salida: 8
```

Código 20: Operación de un exponente

4.3.1. Operadores de Comparación

- `>` : Verifica si el primer operando es mayor que el segundo.

```
    resultado = 10 > 5
    print(resultado)  # Salida: True
```

Código 21: Verificación con operador mayor que

- `>=` : Verifica si el primer operando es mayor o igual que el segundo.

```
    resultado = 10 >= 10
    print(resultado)  # Salida: True
```

Código 22: Verificación con operador mayor o igual que

- `<` : Verifica si el primer operando es menor que el segundo.

```
    resultado = 3 < 8
    print(resultado)  # Salida: True
```

Código 23: Verificación con operador menor que

- `<=` : Verifica si el primer operando es menor o igual que el segundo.

```
    resultado = 5 <= 5
    print(resultado)  # Salida: True
```

Código 24: Verificación con operador menor o igual que

- `==` : Verifica si ambos operandos son iguales.

```
    resultado = 5 == 5
    print(resultado)  # Salida: True
```

Código 25: Verificación con operador igual a

- `!=` : Verifica si ambos operandos no son iguales.

```
    resultado = 5 != 3
    print(resultado)  # Salida: True
```

Código 26: Verificación con operador distinto de

4.3.2. Logica Proposicional

La lógica proposicional en la programación se utiliza para combinar y evaluar proposiciones utilizando operadores lógicos. Mayormente utilizado con las condicionales `if`, `elif` y `else` o también con los bucles `while` y `for`.

- **and:** Devuelve `True` si ambas proposiciones son verdaderas.

```
    resultado = (5 > 2) and (10 < 20)
    print(resultado)  # Salida: True
```

Código 27: Utilización del operador lógico and

- **or:** Devuelve `True` si al menos una de las proposiciones es verdadera.

```
    resultado = (5 < 2) or (10 < 20)
    print(resultado)  # Salida: True
```

Código 28: Utilización del operador lógico or

- **not:** Invierte el valor de verdad de la proposición.

```
    resultado = not (5 > 2)
    print(resultado)  # Salida: False
```

Código 29: Utilización del operador lógico not

Ejemplos de Utilizacion de and, or, not

- Utilización del operador `and` para comprobar si un número es positivo y menor que 10.

```
    numero = 5

    if numero > 0 and numero < 10:
        print(f"{numero} es positivo y menor que 10.")
    else:
        print(f"{numero} no cumple la condicion.")
```

Código 30: Comprobación con el operador lógico and

- Utilización del operador `or` para verificar si un número es negativo o mayor que 20.

```
    numero = 25

    if numero < 0 or numero > 20:
        print(f"{numero} es negativo o mayor que 20.")
    else:
        print(f"{numero} no cumple la condicion.")
```

Código 31: Comprobación con el operador lógico or

- Utilización del operador `not` para invertir la condición de una proposición.

```
numero = 15

if not numero % 2 == 0:
    print(f"{numero} es impar.")
else:
    print(f"{numero} es par.")
```

Código 32: Comprobación con el operador lógico not

- Utilización de más de un operador lógico para comprobar una condición.

```
# Definimos una lista de numeros
numeros = [1, 5, 8, 12, 15, 22]

# Recorremos la lista de numeros
for numero in numeros:
# Condiciones usando and, or, y not
if (numero > 5 and numero < 20) or not (numero % 2 == 0):
    print(f"El numero {numero} cumple con la condicion.")
else:
    print(f"El numero {numero} no cumple con la condicion."
```

Código 34: Ejemplo de operadores lógicos en una condición

4.4. Entrada

Existen dos métodos principales para la entrada de datos en un sistema: la asignación de valores y el ingreso mediante hardware periférico, como el teclado.

4.4.1. Ingreso por asignación

Este método implica la asignación directa de valores a las variables dentro del código del programa. Es una forma estática de introducir datos, que no requiere interacción del usuario durante la ejecución del programa. Este tipo de entrada es común en pruebas y desarrollo, donde los valores predefinidos permiten validar el comportamiento del sistema sin necesidad de interacción en tiempo real.

```
# Ingreso por asignacion
numero_par = 2
numero_decimal = 3.25
animal_domestico = 'perro'
alumnos_fisi = ['Piero', 'Sebastian', 'Renzo', 'christopher']
```

Código 35: Declaraciones por asignación

4.4.2. Ingreso por hardware de entrada

El ingreso por hardware se refiere a la introducción de datos a través de dispositivos periféricos, como el teclado, el mouse, o dispositivos de entrada más especializados como escáneres, micrófonos o cámaras. Estos dispositivos permiten la interacción dinámica del usuario con el sistema, enviando datos en tiempo real.

Ejemplos de dispositivos:

- **Teclado:** Permite la entrada de texto y comandos. Es uno de los periféricos de entrada más comunes.
- **Mouse:** Utilizado para navegar y seleccionar elementos en la interfaz gráfica del usuario.
- **Escáner:** Permite digitalizar documentos o imágenes físicas para su procesamiento en el sistema.
- **Micrófono:** Captura sonido, lo que puede ser útil para aplicaciones de reconocimiento de voz o grabación de audio.
- **Cámara:** Capturando imágenes, utilizado mayormente para procesamiento de imágenes en tiempo real.

El ingreso por hardware a menudo requiere controladores específicos y puede depender del sistema operativo para interpretar las señales del dispositivo. Además, la calidad de los datos ingresados puede variar dependiendo del dispositivo y de su configuración.

input: La función input() se utiliza para leer datos de entrada del usuario desde la consola. Devuelve la entrada del usuario como una cadena de texto.

```
1    #Utilizacion de un input para ingresar un valor y asignarle a una variable.
2    nombre = input("Introduce tu nombre: ")
3    print(f"Hola, {nombre}!")
```

Código 36: Declaraciones por asignación

4.5. Tipos de Nomenclatura

En Python, existen varias convenciones para nombrar variables[6] y otros identificadores. Estas convenciones ayudan a que el código sea más legible y mantenible.

4.5.1. Snake Case

Este es el estilo preferido en Python para nombrar variables y funciones. Utiliza guiones bajos para separar las palabras. Mayormente utilizado para declarar variables en Python, este método ayuda a describir el propósito de la variable.

```
1    nombre_completo = "Juan Perez"
2    puntaje_total = 1500
3    numero_de_intentos = 3
4    nombre_usuario = "Maria"
5    nota_parcial = 17
6    codigo_estudiante = 29011245
```

Código 37: Declaración de variables en Snake Case

[6]Cuando una variable se utiliza en un contexto en el que su valor no es relevante o su único propósito es cumplir con una sintaxis específica, se estandariza utilizar un guion bajo _ como nombre de la variable. Esto indica que el valor de la variable no se utilizará y ayuda a evitar la necesidad de buscar un nombre significativo para ella.

4.5.2. Camel Case

Comienza con una letra minúscula y cada palabra subsiguiente empieza con una letra mayúscula. Este estilo no es comúnmente utilizado para variables en Python, pero a veces se usa en nombres de funciones o métodos en bibliotecas externas.

```
    nombreCompleto = "Juan Perez"
    edadEstudiante = 20
```

Código 38: Declaración de variables en Camel Case

4.5.3. Pascal Case (Upper Camel Case)

Similar a Camel Case, pero la primera letra también es mayúscula. Este estilo se utiliza a menudo para los nombres de clases en Python. Esto ayuda a diferenciar las clases de las variables y funciones.

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

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

Código 39: Declaración de clases

4.5.4. Upper Case (Mayúsculas y guiones bajos)

Las constantes se utilizan para valores que no deberían cambiar durante la ejecución del programa. Se nombran usando Upper Case.

```
    PI = 3.14159
    MAX_CONNECTIONS = 100
    TASA_DE_INTERES = 0.05
    MAX_VELOCIDAD = 120
```

Código 40: Declaración de constantes en Upper Case

Existen diferentes convenciones para nombrar variables, y estas varían según el lenguaje de programación que se esté utilizando. Las prácticas de nombramiento de variables que se han mostrado en estos casos son consideradas buenas prácticas de programación[7].

4.6. Salida

Para mostrar información en pantalla en Python, comúnmente utilizamos el método `print()`. Python ofrece varias técnicas para formatear cadenas, permitiendo insertar variables y controlar aspectos como espaciado, alineación y precisión de los datos. A continuación, se describen las principales técnicas de formateo de cadenas disponibles en Python:

[7]Conjunto de principios y técnicas que los desarrolladores siguen para escribir código que sea más fácil de entender, mantener y mejorar. Estas prácticas buscan mejorar la calidad del software y facilitar su desarrollo a largo plazo.

4.6.1. Operador % (Porcentaje)

También conocido como "interpolación de cadenas", este método clásico utiliza marcadores de posición en la cadena que se reemplazan con valores proporcionados. Los marcadores de posición comunes incluyen:

- **%s**: Para cadenas de texto.
- **%d**: Para enteros.
- **%f**: Para números de punto flotante.

```
nombre = "Juan"
edad = 30
altura = 1.75
print("Nombre: %s, Edad: %d, Altura: %.2f" % (nombre, edad, altura))

#En este ejemplo, nombre se inserta en el lugar de %s, edad en %d, y altura en
 %f con dos decimales.
```

Código 41: Utilización de print con Operador %

4.6.2. Método format()

El método `format()` proporciona una mayor flexibilidad y claridad. Utiliza llaves como marcadores de posición dentro de la cadena. Puedes especificar el formato directamente dentro de las llaves.

```
nombre = "Juan"
edad = 30
altura = 1.75
print("Nombre: {}, Edad: {}, Altura: {:.2f}".format(nombre, edad, altura))

#Aqui, las llaves {} se reemplazan por los valores proporcionados en el metodo
 format().
#{:.2f} indica que el numero de punto flotante debe mostrarse con dos
decimales.
```

Código 42: Utilización de print con método format

4.6.3. F-Strings (Interpolación de Cadenas Literal)

Los F-Strings[8] permiten una forma concisa y legible de incrustar expresiones dentro de literales de cadena. Se utilizan precediendo la cadena con la letra `f` y colocando las variables o expresiones dentro de llaves.

```
nombre = "Juan"
edad = 30
altura = 1.75
print(f"Nombre: {nombre}, Edad: {edad}, Altura: {altura:.2f}")

#En este caso, "f" antes de las comillas indica que se esta utilizando una F-
String. Las variables nombre, edad y altura se insertan directamente en la
cadena, con {altura:.2f} para limitar el numero de decimales a dos.
```

Código 43: Utilización de print con F-Strings

[8] Mayormente utilizado por su facilidad de sintaxis y manejo de las variables al momento de salida por consola

4.7. Condicionales

Las declaraciones condicionales permiten a los programas tomar decisiones y ejecutar diferentes bloques de código basados en condiciones.

4.7.1. if, elif, else

Las estructuras de control `if`, `elif` y `else` se utilizan para ejecutar bloques de codigo condicionalmente. La condicion se evalua y, si es verdadera, se ejecuta el bloque de codigo correspondiente.

```
x = 10

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

Código 44: Utilización de if, elif y else

Casos Comunes de Uso de `if, elif, else`

- **Verificación de Condiciones:** Ejecutar diferentes bloques de código según se cumplan ciertas condiciones.
- **Toma de Decisiones:** Realizar diferentes acciones en función del valor de una variable o el resultado de una expresión.

Ejemplos de Utilización de if, elif, else

- Utilización de **if, elif, else** para verificar si un número es negativo, positivo o cero.

```
numero = -5

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

Código 45: Verificar si un número es negativo, positivo o cero.

- Utilización de condicionales para asignar una calificación basada en una puntuación.

```
puntuacion = 85

if puntuacion >= 90:
    calificacion = "A"
elif puntuacion >= 80:
    calificacion = "B"
elif puntuacion >= 70:
    calificacion = "C"
elif puntuacion >= 60:
    calificacion = "D"
```

```
    else:
        calificacion = "F"

    print(f"La calificacion es: {calificacion}")
```

Código 46: Asignar una calificación basada en una puntuación.

- Utilización de condicionales para determinar el tipo de un triángulo según la longitud de sus lados.

```
    lado1 = 5
    lado2 = 5
    lado3 = 5

    if lado1 == lado2 == lado3:
        tipo = "Equilatero"
    elif lado1 == lado2 or lado2 == lado3 or lado1 == lado3:
        tipo = "Isosceles"
    else:
        tipo = "Escaleno"

    print(f"El triangulo es: {tipo}")
```

Código 47: Determinar el tipo de triángulo según la longitud de sus lados.

- Utilización de condicionales para decidir una acción basada en la edad de una persona.

```
    edad = 20

    if edad < 18:
        print("Eres menor de edad")
    elif 18 <= edad < 65:
        print("Eres un adulto")
    else:
        print("Eres un adulto mayor")
```

Código 48: Decidir una acción basada en la edad de una persona.

- Utilización de condicionales para determinar si un número es par o impar.

```
    numero = 42

    if numero % 2 == 0:
        print(f"El numero {numero} es par")
    else:
        print(f"El numero {numero} es impar")
```

Código 49: Determinar si un número es par o impar con `if`, `else`

4.7.2. match

La declaración `match` en Python (disponible a partir de Python 3.10) se utiliza para realizar comparaciones estructurales, similar a la declaración `switch` en otros lenguajes de programación.

```
    comando = "iniciar"

    match comando:
        case "iniciar":
            print("Iniciando el programa...")
        case "detener":
            print("Deteniendo el programa...")
        case _:
            print("Comando no reconocido.")

    # En este ejemplo, se utiliza la declaracion match para comparar el valor de
    una variable y ejecutar el codigo correspondiente.
```

Código 50: Uso de la declaración match

4.7.3. Excepciones

Las excepciones en Python son eventos que ocurren durante la ejecución de un programa y que interrumpen el flujo normal de las instrucciones. Se utilizan para manejar errores y situaciones excepcionales que pueden surgir durante la ejecución del código.

```
    try:
        resultado = 10 / 0
    except ZeroDivisionError:
        print("Error: Division por cero")
```

Código 51: Manejo de una excepción de división por cero

Casos Comunes de Uso de try-except

- **Manejo de Errores de Ejecución:** Capturar y manejar errores que ocurren durante la ejecución del código, evitando que el programa se detenga abruptamente.
- **Validación de Datos:** Comprobar y manejar situaciones donde los datos de entrada puedan causar errores en el programa.

Ejemplos de Utilización de Excepciones

- Manejo de una excepción para capturar errores de división por cero.

```
    try:
        divisor = int(input("Introduce un numero para dividir: "))
        resultado = 10 / divisor
    except ZeroDivisionError:
        print("Error: No se puede dividir por cero.")
    except ValueError:
        print("Error: Introduce un valor numerico valido.")
    else:
        print(f"El resultado es: {resultado}")
    #El else se cumple si se llega a la excepcion de error.
```

Código 52: Manejo de división por cero y valor no numérico

- Manejo de una excepción para abrir un archivo que no existe.

```
    try:
        with open("archivo_inexistente.txt", "r") as archivo:
            contenido = archivo.read()
    except FileNotFoundError:
        print("Error: El archivo no se encontro.")
    #Podria darse el caso de que no encontremos el archivo o otro tipo de
    documento, para ellos utilizamos esta excepcion para evitar terminar el
    programa por el error.
```

Código 53: Manejo de archivo no encontrado

- Utilización de una cláusula `finally` para ejecutar código que debe correr sin importar si ocurre una excepción o no.

```
    try:
        archivo = open("archivo.txt", "r")
        contenido = archivo.read()
    except FileNotFoundError:
        print("Error: El archivo no se encontro.")
    finally:
        archivo.close()
        print("El archivo ha sido cerrado.")
    #El finally siempre se ejecuta, seria como la continuacion del codigo,
    aunque tambien se puede obviar y continuar con el codigo
```

Código 54: Uso de `finally` para cerrar un archivo

- Captura de múltiples excepciones en una sola cláusula `except`.

```
    try:
        valor = int(input("Introduce un numero: "))
        resultado = 10 / valor
    except (ZeroDivisionError, ValueError) as error:
        print(f"Error: {error}")
    #Se puede utilizar mas de una excepcion en un except, la recomendacion es
     seguir la estructura que se presenta en el ejemplo para evitar errores
    de compilacion
```

Código 55: Captura de múltiples excepciones

- Manejo de errores al trabajar con la red, como la conexión a un servidor.

```
    import requests

    try:
        respuesta = requests.get("https://www.example.com")
        respuesta.raise_for_status()
    except requests.exceptions.RequestException as e:
        print(f"Error de red: {e}")
    #Existen casos como una falla de conexion a internet o que la pagina web
    a la que se esta tratando de hacer la solicitud get ya no exista.
```

Código 56: Manejo de errores de red

- Manejo de errores de conversión, como convertir una cadena a un número.

```
    cadena = "123a"

    try:
        numero = int(cadena)
    except ValueError:
        print("Error: La cadena no se puede convertir a un numero entero.")
    #Los casos en los que no se tiene idea de que tipo de variable estamos
    guardando y aun deseamos convertirlo en entero o float, pero en esa
    asignacion encontramos una cadena, para evitar el error por cambio de
    naturaleza de variable, utilizamos esta excepcion.
```

Código 57: Manejo de error de conversión

- Manejo de excepciones al acceder a claves en un diccionario que no existen.

```
    diccionario = {"clave1": 1, "clave2": 2}

    try:
        valor = diccionario["clave_inexistente"]
    except KeyError:
        print("Error: La clave no existe en el diccionario.")
    #Cuando tratamos de acceder a una clave no existente dentro de un
    diccionario.
```

Código 58: Manejo de error de clave en un diccionario

4.8. Bucles

Los bucles permiten ejecutar un bloque de código repetidamente mientras se cumple una condición o sobre una secuencia de elementos.

4.8.1. while

El bucle `while` ejecuta un bloque de código mientras una condición es verdadera.

```
    contador = 0
    while contador < 5:
        print("Contador:", contador)
        contador += 1
```

Código 59: Utilización de un While

Casos Comunes de Uso de while

- **Esperar un Evento:** Mantener la ejecución de un programa a la espera de que ocurra un evento específico.
- **Leer Datos hasta una Condición:** Leer datos de una fuente hasta que se cumpla una condición particular.

Ejemplos de Utilización de `while`

- Utilización un bucle `while` para esperar a que un usuario introduzca una contraseña correcta.

```
    true_password = "python123"
    entrada = ""

    while entrada != true_password:
        entrada = input("Introduce el password: ")
        if entrada != true_password:
            print("Incorrect password. Intentalo de nuevo.")

    print("Password True")
```

Código 60: Verificación de contraseña con `while`

- Utilización un bucle `while` para buscar un número en una lista.

```
    numeros = [1, 3, 5, 7, 9, 11, 13, 15]
    buscar = 7
    indice = 0
    encontrado = False

    while indice < len(numeros) and not encontrado:
        if numeros[indice] == buscar:
            encontrado = True
        else:
            indice += 1

    if encontrado:
        print(f"Numero {buscar} encontrado en el indice {indice}")
    else:
        print(f"Numero {buscar} no encontrado en la lista")
```

Código 61: Búsqueda de un número en una lista con `while`

- Utilización un bucle `while` para calcular la suma de los dígitos de un número.

```
    numero = 12345
    suma_digitos = 0

    while numero > 0:
        digito = numero % 10
        suma_digitos += digito
        numero = numero // 10

    print(f"La suma de los digitos es: {suma_digitos}")
```

Código 62: Cálculo de la suma de los dígitos de un número con `while`

- Utilización un bucle `while` para generar los primeros n números de Fibonacci.

```
    n = 10
    a, b = 0, 1
    contador = 0

    while contador < n:
        print(a, end=" ")
        a, b = b, a + b
        contador += 1
    print()
```

Código 63: Generación de números de Fibonacci con `while`

4.8.2. for

El bucle `for` itera sobre una secuencia (como una lista, tupla, diccionario, conjunto o cadena) y ejecuta un bloque de código para cada elemento en la secuencia.

```
    for contador in range(5):
        print("Contador:", contador)
```

Código 64: Utilización de un For

Casos Comunes de Uso de for

- **Iterar sobre una Secuencia:** Ejecutar un bloque de código para cada elemento de una secuencia (lista, tupla, rango, etc.).
- **Procesar Elementos de un Conjunto:** Realizar operaciones sobre cada elemento de un conjunto de datos.

Ejemplos de Utilización de `for`

- Utilización de un bucle `for` para iterar sobre una lista de nombres [9] e imprimir un saludo personalizado.

```
    nombres = ["Alice", "Bob", "Charlie", "Diana"]

    for nombre in nombres:
        print(f"Hola, {nombre}!")
```

Código 65: Saludo personalizado con `for`

- Utilización de un bucle `for` para calcular la suma de todos los elementos en una lista.

```
    numeros = [1, 2, 3, 4, 5]
    suma = 0

    for numero in numeros:
        suma += numero

    print(f"La suma de los numeros es: {suma}")
```

Código 66: Suma de elementos en una lista con `for`

- Utilización de un bucle `for` para generar los primeros n números de la sucesión de Fibonacci.

```
    n = 10
    a, b = 0, 1 #Declaracion por separacion de ,
    #For utilizando un varialble iterativa temporal
    for _ in range(n):
        print(a, end=" ")
        a, b = b, a + b
    print()
```

[9] La palabra nombre se está utilizando para iterar sobre nombres, pero también puede cualquier otra palabra para ese iterable.

Código 67: Generación de números de Fibonacci con `for`

- Utilización de un bucle `for` para recorrer una cadena y contar las vocales.

```
    cadena = "Hola, buenas tardes"
    vocales = "aeiouAEIOU"
    contador_vocales = 0

    for letra in cadena:
        if letra in vocales:
            contador_vocales += 1

    print(f"El numero de vocales en la cadena es: {contador_vocales}")
```

Código 68: Contar vocales en una cadena con `for`

4.8.3. break

La instrucción `break` se utiliza para salir de un bucle antes de que se complete su ejecución normal.

```
    for numero in range(10):
        if numero == 5:
            break
        print("Numero:", numero)
    # El print("Numero", numero) se ejecutaria hasta llegar a 4, porque en 5 se
    sale del bucle por el break en la condicion if numero == 5.
```

Código 69: Utilización de un `break`

4.8.4. continue

La instrucción `continue` se utiliza para saltar a la siguiente iteración de un bucle.

```
    for numero in range(10):
        if numero % 2 == 0:
            continue
        print("Numero impar:", numero)
    #El print("Numero impar:", numero) solo mostraria los numeros impares, debido
    a que dentro de la condicion if numero % 2 == 0 existe un continue, el cual
    hace pasar defrente a la siguiente iteracion, obviando toda accion
    subsiguiente luego de continue, mientras este en el bucle.
```

Código 70: Utilización de un `continue`

4.8.5. pass

La instrucción `pass` no realiza ninguna acción y se utiliza como un marcador de posición cuando se requiere una declaración sintácticamente, pero no se necesita ninguna acción.

```
    for numero in range(5):
        if numero == 3:
            pass  # Aqui no se realiza ninguna accion
        else:
            print("Numero:", numero)
    #La instruccion pass se utiliza para no realizar ninguna accion dentro de una
    condicion, bucle, etc. Es para no dar fallos en un codigo si es que tienes
    una condicion pero no quieres realizar nada en esa condicion. En el ejemplo
    de arriba se mostraria los numeros del 1 al 5, excepto el 3.
```

Código 71: Utilización de un `pass`

4.8.6. Iterable y Variable Iterativa

- Un iterable es cualquier objeto en Python que puede ser recorrido o iterado.

```
    #Iteracion sobre una lista de frutas
    frutas = ["manzana", "banana", "cereza"]
    for fruta in frutas:
        print(fruta)

    #Iteracion sobre una cadena
    palabra = "hola"
    for letra in palabra:
        print(letra)

    #Iteracion sobre un diccionario con datos de una persona
    persona = {"nombre": "Juan", "edad": 30}
    for clave in persona:
        print(clave, persona[clave])

    #Las estructuras de datos son objetos iterable
```

Código 72: Ejemplos de un Iterable

- Una variable iterativa es una variable que toma el valor de cada elemento del iterable en cada iteración del bucle.

```
    # Definimos un iterable
    numeros = [10, 20, 30, 40]

    # Iteramos sobre el iterable usando una variable iterativa
    for numero in numeros:
        cuadrado = numero ** 2
        print(f"El cuadrado de {numero} es {cuadrado}")
```

Código 73: Ejemplos de un Iterable y una Variable Iterativa

4.8.7. Variable Acumulativa

Un acumulador es una variable que se utiliza para reunir, sumar o combinar valores de una secuencia durante la iteración. Los acumuladores son comunes en operaciones que implican suma, conteo, concatenación u otra forma de agregación de datos.

```
    #El acumulador de nombre suma, guarda la suma de los numeros dentro de la
    lista.
    numeros = [1, 2, 3, 4, 5]
    suma = 0
    for numero in numeros:
        suma += numero
    print("Suma:", suma)

    #Cuenta la cantidad de veces que se repite una palabra en una oracion.
    texto = "hola mundo hola"
    conteo_palabras = {}
    for palabra in texto.split():
        if palabra in conteo_palabras:
            conteo_palabras[palabra] += 1
        else:
            conteo_palabras[palabra] = 1
    print("Conteo de palabras:", conteo_palabras)
```

Código 74: Ejemplos de un Variable Acumulativa

4.9. Funciones

Las funciones en Python permiten agrupar un conjunto de instrucciones bajo un nombre, facilitando su reutilización y organización. Se definen usando la palabra clave `def` y pueden aceptar parámetros y devolver resultados.

```
    def nombre_funcion(parametro1, parametro2):
        # Codigo que realiza una tarea

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

    saludar("Alice")
```

Código 75: Definición y llamada de una función simple

4.9.1. Paradigma Modular

El paradigma[10] modular es una metodología de desarrollo de software que enfatiza la división del programa en partes más pequeñas y manejables llamadas módulos. Cada módulo debe ser una unidad autónoma que encapsula una parte específica de la funcionalidad del programa. Esta división facilita la comprensión, el desarrollo, la prueba y el mantenimiento del software.

- **División del Trabajo:** Divide el programa en módulos más pequeños y específicos, como funciones, clases o paquetes.
- **Reusabilidad:** Los módulos bien diseñados pueden ser reutilizados en diferentes partes del programa o en otros proyectos.
- **Mantenibilidad:** Facilita la localización y corrección de errores, así como la implementación de mejoras.
- **Encapsulamiento:** Cada módulo debe ocultar su implementación interna y exponer solo lo necesario para su uso. Esto promueve la separación de preocupaciones.

```
    def leer_datos():
        # Simulacion de lectura de datos
        return [1, 2, 3, 4, 5]

    def procesar_datos(datos):
        # Procesamiento simple: calcular el cuadrado de cada numero
        return [dato ** 2 for dato in datos]

    def mostrar_resultados(resultados):
        # Mostrar los resultados procesados
        for resultado in resultados:
            print(resultado)

    # Programa principal
    datos = leer_datos()
    resultados = procesar_datos(datos)
    mostrar_resultados(resultados)
```

Código 76: Ejemplo del paradigma modular

[10] Estilo o enfoque de programación que proporciona un marco conceptual y una metodología para diseñar y escribir programas. Los paradigmas de programación definen la manera en que se estructuran y organizan los programas, así como las técnicas y conceptos que se utilizan para resolver problemas y manipular datos.

4.9.2. Parámetros en una Funcion

- **Parámetros Posicionales:** Son los parámetros estándar que se pasan a una función en un orden específico.

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

    resultado = multiplicar(4, 5)
    print(f"El resultado de multiplicar 4 y 5 es: {resultado}")
```

Código 77: Ejemplo de parámetros posicionales

- **Parámetros con Valor Predeterminado:** Son parámetros que tienen un valor por defecto y pueden ser omitidos al llamar a la función.

```
    def saludar(nombre, mensaje="Hola"):
    return f"{mensaje}, {nombre}"

    print(saludar("Juan"))  # Hola, Juan
    print(saludar("Juan", "Buenos dias"))  # Buenos dias, Juan
```

Código 78: Ejemplo de parámetro con valor predeterminado

- **Parámetros con *args:** Permite pasar un número variable de argumentos posicionales a una función. Dentro de la función, `args` se convierte en una tupla que contiene todos los argumentos posicionales adicionales.

```
    def sumar_todos(*args):
    return sum(args)

    print(sumar_todos(1, 2, 3))  # 6
    print(sumar_todos(4, 5, 6, 7, 8))  # 30
```

Código 79: Ejemplo de parámetro con *args

- **Parámetros con **kwargs:** Permite pasar un número variable de argumentos con nombre (keyword arguments) a una función. Dentro de la función, kwargs se convierte en un diccionario que contiene todos los argumentos con nombre adicionales.

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

    mostrar_info(nombre="Juan", edad=30, ciudad="Lima")
    # nombre: Juan
    # edad: 30
    # ciudad: Lima
```

Código 80: Ejemplo de parámetro con **kwargs

```python
def combinar(a, b, *args, **kwargs):
print(f"a: {a}")
print(f"b: {b}")
print("args:", args)
print("kwargs:", kwargs)

combinar(1, 2, 3, 4, 5, x=10, y=20)
# a: 1
# b: 2
# args: (3, 4, 5)
# kwargs: {'x': 10, 'y': 20}
```

Código 81: Ejemplo de combinación de parámetros

4.9.3. Funciones Lambda

Las funciones lambda son funciones anónimas que se definen con la palabra clave `lambda`. Son útiles para operaciones simples que se utilizan una sola vez o para funciones pequeñas que se pasan como argumentos a otras funciones.

```python
lambda argumentos: expresion
```

Código 82: Ejemplo de una función `lambda`

Ejemplos de Funciones Lambda

- Función lambda para calcular el doble de un número.

```python
#Se esta utilizando una funcion anonima lambda para obtener x elevado al cuadrado, sin tener que crear una funcion y pasar por parametros el x.
doble = lambda x: x * 2
print(f"El doble de 6 es: {doble(6)}")

#Salida <El doble de 6 es: 12>
```

Código 83: Función lambda para calcular el doble

- Utilización de `map()` con una función lambda para aplicar una operación a cada elemento de una lista.

```python
#Se esta utilizando un map para iterar por cada valor de la lista numeros, aplicando una elevacion al cuadrado, ademas se esta aplicando list para que dobles sea una lista.
numeros = [1, 2, 3, 4]
dobles = list(map(lambda x: x * 2, numeros))
print(f"Dobles de la lista: {dobles}")

#Salida <Dobles de la lista: [2, 4, 6, 8]>
```

Código 84: Uso de `map()` con lambda

- Utilización de `filter()` con una función lambda para filtrar elementos de una lista.

```
    #Se esta utilizando filter para filtar solo los numeros pares dentro de
    la lista numeros, ademas se esta utiliznado list para convertir en una
    lista la variable pares.
    numeros = [1, 2, 3, 4, 5, 6]
    pares = list(filter(lambda x: x % 2 == 0, numeros))
    print(f"Numeros pares de la lista: {pares}")

    #Salida <Numeros pares de la lista: [2, 4, 6]>
```

Código 85: Uso de `filter()` con lambda

- Utilización de `sorted()` con una función lambda para ordenar una lista de tuplas por el segundo elemento.

```
    #Se esta utilizando sorterd para ordenar la tupla desde su segundo
    elemento, en este caso alfabeticamente
    tuplas = [(1, 'uno'), (3, 'tres'), (2, 'dos')]
    ordenado = sorted(tuplas, key=lambda x: x[1])
    print(f"Tuplas ordenadas por el segundo elemento: {ordenado}")

#Salida
#<Tuplas ordenadas por el segundo elemento: [(2,'dos'),(1,'uno'),(3,'tres')]>
```

Código 86: Uso de `sorted()` con lambda

4.9.4. Funciones recursivas

Las funciones recursivas son aquellas que se llaman a sí mismas durante su ejecución. La recursión es una técnica útil para problemas que pueden dividirse en subproblemas similares. Una función recursiva debe tener dos componentes clave:

- **Caso base**: Una condición que detiene la recursión.
- **Llamada recursiva**: La función se llama a sí misma con un subconjunto del problema original.
- El cálculo del factorial de un número es un ejemplo clásico de recursión. El factorial de un número n (denotado como $n!$) se define como el producto de todos los enteros positivos menores o iguales a n. El caso base es cuando $n = 0$, donde $0! = 1$.

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

    resultado = factorial(5)
    print(f"El factorial de 5 es {resultado}")
    #En este ejemplo, factorial calcula el factorial de un numero n llamando
    a si misma con n - 1 hasta llegar al caso base.
```

Código 87: Método 1 de recursividad para calcular el factorial

- Aplicación común de la recursión es la serie de Fibonacci, donde cada número es la suma de los dos números anteriores. La serie comienza con 0 y 1.

```
    def fibonacci(n):
        if n <= 1:
            return n
        else:
            return fibonacci(n - 1) + fibonacci(n - 2)

    resultado = fibonacci(6)
    print(f"El numero Fibonacci en la posicion 6 es {resultado}")
    #En este ejemplo, fibonacci calcula el enesimo numero de la serie de
    Fibonacci. La funcion se llama a si misma dos veces: una para n - 1 y
    otra para n - 2, hasta llegar a los casos base.
```

Código 88: Método 2 de recursividad para calcular el factorial

- Aunque la recursión puede ser simplificar el código, también puede llevar a problemas como el desbordamiento de pila (stack overflow) si la profundidad de la recursión es muy grande. En tales casos, las soluciones iterativas pueden ser más eficientes.

```
    def cuenta_regresiva(n):
        if n < 0:
            return
        print(n)
        cuenta_regresiva(n - 1)

    cuenta_regresiva(5)
    #Este ejemplo muestra una funcion recursiva simple que cuenta hacia atras
     desde un numero dado hasta 0. La condicion de parada es cuando n es
    menor que 0, pero si no hubiera caso base o que este mal planteado la
    condicion base se produciria desbordamiento.
```

Código 89: Ejemplo de un buen planteamiento de recursividad

4.9.5. Funciones Built-in

Las funciones built-in son aquellas que Python proporciona por defecto y que puedes usar directamente en tu código. Algunas de las más comunes incluyen:

- **Funciones Matemáticas:**

```
    #abs() - Devuelve el valor absoluto de un numero.
    print(abs(-5))  # Salida: 5
    print(abs(3.14))  # Salida: 3.14

    #round() - Redondea un numero a un numero especifico de decimales.
    print(round(3.14159, 2))  # Salida: 3.14
    print(round(2.71828))  # Salida: 3

    #min() - Devuelve el valor minimo de un conjunto de valores.
    print(min(1, 2, 3, 4, 5))  # Salida: 1
    print(min([10, 20, 5, 15]))  # Salida: 5

    #max() - Devuelve el valor maximo de un conjunto de valores.
    print(max(1, 2, 3, 4, 5))  # Salida: 5
    print(max([10, 20, 5, 15]))  # Salida: 20

    #sum() - Devuelve la suma de un iterable (como una lista).
    print(sum([1, 2, 3, 4, 5]))  # Salida: 15
    print(sum((10, 20, 30)))  # Salida: 60

    #divmod() - Devuelve una tupla que contiene el cociente y el residuo de
    la division de dos numeros.
```

```
    print(divmod(9, 4))  # Salida: (2, 1)
    print(divmod(10, 3))  # Salida: (3, 1)

    #pow() - Devuelve el resultado de elevar un numero a una potencia, o de
    elevar un numero a una potencia y luego tomar el modulo de ese resultado.
    print(pow(2, 3))  # Salida: 8
    print(pow(2, 3, 5))  # Salida: 3 (8 mod 5)
```

Código 90: Utilización de funciones matemáticas

- **Funciones de Conversión:** Más información en la subsección 4.1
- **Funciones de Secuencia:**

```
    #len() - Devuelve la longitud de una secuencia (el numero de elementos).
    lista = [1, 2, 3, 4]
    print(len(lista))  # Salida: 4

    cadena = "Hola"
    print(len(cadena))  # Salida: 4

    #enumerate() - Devuelve un iterador que produce pares de indice y valor
    para cada elemento de una secuencia.
    lista = ['a', 'b', 'c']
    for indice, valor in enumerate(lista):
        print(f"Indice: {indice}, Valor: {valor}")
    # Salida:
    # Indice: 0, Valor: a
    # Indice: 1, Valor: b
    # Indice: 2, Valor: c

    #zip() - Combina multiples secuencias en una sola secuencia de tuplas,
    donde cada tupla contiene un elemento de cada secuencia.
    nombres = ['Juan', 'Ana', 'Luis']
    edades = [28, 22, 35]
    for nombre, edad in zip(nombres, edades):
        print(f"{nombre} tiene {edad} anios")
    # Salida:
    # Juan tiene 28 anios
    # Ana tiene 22 anios
    # Luis tiene 35 anios

    #sorted() - Devuelve una nueva lista con los elementos de la secuencia
    ordenados.
    lista = [3, 1, 4, 1, 5, 9]
    print(sorted(lista))  # Salida: [1, 1, 3, 4, 5, 9]

    cadena = "python"
    print(sorted(cadena))  # Salida: ['h', 'n', 'o', 'p', 't', 'y']

    #reversed() - Devuelve un iterador que produce los elementos de la
    secuencia en orden inverso.
    lista = [1, 2, 3, 4]
    print(list(reversed(lista)))  # Salida: [4, 3, 2, 1]

    cadena = "abc"
    print(''.join(reversed(cadena)))  # Salida: 'cba'
```

Código 91: Utilización de funciones Secuenciales

- **Funciones de Iteración:**

```python
# Usando range con un solo argumento (stop)
for i in range(5):
    print(i)
# Salida:
# 0
# 1
# 2
# 3
# 4

# Usando range con start y stop
for i in range(2, 5):
    print(i)
# Salida:
# 2
# 3
# 4

# Usando range con start, stop y step
for i in range(1, 10, 2):
    print(i)
# Salida:
# 1
# 3
# 5
# 7
# 9
#Nota: Siempre es uno menos al limite superior.
```

Código 92: Utilización de funciones iterativas

- **Funciones de Tipo:**

```python
#type(objeto): Devuelve el tipo del objeto.
print(type(123))      # Salida: <class 'int'>
print(type("texto"))  # Salida: <class 'str'>

#isinstance(objeto, tipo): Verifica si el objeto es una instancia del
tipo especificado o de una subclase de este tipo.
print(isinstance(123, int))      # Salida: True
print(isinstance("texto", str))  # Salida: True
print(isinstance(123, str))      # Salida: False
```

Código 93: Ejemplo de un buen planteamiento de recursividad

- **Funciones de Entrada/Salida:**

```python
print("Hola, mundo!")                  # Salida: Hola, mundo!
print("Numero:", 42)                   # Salida: Numero: 42
print("Lista:", [1, 2, 3])             # Salida: Lista: [1, 2, 3]
print("Valor:", 3.14, sep="--")        # Salida: Valor:--3.14
```

Código 94: Ejemplo de un buen planteamiento de recursividad

Más información sobre las funciones de entrada en subsección 4.6 y la subsección 5.4.

5. Python Intermedio

5.1. Código Pythonico

El código pythonico se refiere a escribir código en un estilo que sea claro, conciso y utilice las características y convenciones idiomáticas de Python. El objetivo es escribir código que no solo funcione, sino que también sea fácil de leer y mantener.

Tipos de Código Pythonico

- **Comprensiones de Listas:** Permiten crear listas de manera concisa y legible. Se utilizan para aplicar una operación a cada elemento de una secuencia y filtrar los elementos que cumplen una condición.

```
    # Comprension de lista para obtener cuadrados de numeros pares
    numeros = [1, 2, 3, 4, 5]
    cuadrados = [n**2 for n in numeros if n % 2 == 0]
    print(cuadrados)  # Salida: [4, 16]
```

Código 95: Ejemplo de Comprensiones de Listas

- **Expresiones Generadoras:** Son similares a las comprensiones de listas, pero en lugar de crear una lista completa, generan elementos sobre la marcha, lo que puede ser más eficiente en términos de memoria.

```
    # Expresion generadora para obtener cuadrados de numeros pares
    numeros = [1, 2, 3, 4, 5]
    generador = (n**2 for n in numeros if n % 2 == 0)
    print(list(generador))  # Salida: [4, 16]

    # No pythonico
    generador = []

    for n in numeros:
        if n % 2 == 0:
            generador.append(n**2)

    print(generador)  # Salida: [4, 16]
```

Código 96: Ejemplo de Expresiones Generadoras

- **Funciones Lambda:** Permiten crear funciones pequeñas y anónimas en una sola línea. Se utilizan comúnmente con funciones de orden superior como `map`, `filter`, y `sorted`.

```
    #Uso de funcion lambda para obtener cuadrados de numeros pares
    numeros = [1, 2, 3, 4, 5]
    cuadrados = list(map(lambda n: n**2, filter(lambda n: n % 2 == 0, numeros
    )))
    print(cuadrados)  # Salida: [4, 16]

    #No pythonico
    cuadrados = []

    for n in numeros:
        if n % 2 == 0:
            cuadrados.append(n**2)

    print(cuadrados)  # Salida: [4, 16]
```

Código 97: Ejemplo de Funciones Lambda

- **Desempaquetado de Tuplas:** Permite extraer valores de una tupla en varias variables en una sola línea, lo que mejora la claridad del código.

```
    # Desempaquetado de tuplas en una sola linea
    pares = [(1, 'uno'), (2, 'dos'), (3, 'tres')]
    for numero, nombre in pares:
        print(f"{numero}: {nombre}")
        # Salida:
        # 1: uno
        # 2: dos
        # 3: tres
```

Código 98: Ejemplo de código Pythonico

- **Uso de with para Manejo de Recursos:** Garantiza que los recursos se liberen correctamente, como archivos o conexiones, incluso si ocurre un error.

```
    # No pythonico
    file = open('file.txt', 'r')
    content = file.read()
    file.close()

    # Pythonico
    with open('file.txt', 'r') as file:
        content = file.read()
```

Código 99: Ejemplo de Uso de with para Manejo de Recursos

- **Documentación con Docstrings:** Permite documentar funciones, clases y módulos utilizando cadenas de documentación que pueden ser accedidas a través de la función `help()`.

```
    """
    Este modulo proporciona funciones utilitarias para operaciones
    matematicas avanzadas.

    Funciones:
    - suma(a, b): Retorna la suma de dos numeros.
    - resta(a, b): Retorna la resta de dos numeros.
    """

    def suma(a, b):
        """
        Retorna la suma de dos numeros.

        Args:
        a (int, float): El primer numero.
        b (int, float): El segundo numero.

        Returns:
        int, float: La suma de a y b.
        """
        return a + b

    def resta(a, b):
        """
        Retorna la resta de dos numeros.

        Args:
```

```
        a (int, float): El primer numero.
        b (int, float): El segundo numero.

        Returns:
        int, float: La resta de a y b.
        """
        return a - b
```

Código 100: Ejemplo de Documentación con Docstrings

- **Uso de enumerate para Iteraciones:** Permite iterar sobre una secuencia mientras se obtiene el índice de cada elemento.

```
    # Uso de enumerate para iterar con indices
    nombres = ['Ana', 'Luis', 'Carlos']
    for indice, nombre in enumerate(nombres):
        print(f"Indice {indice}: {nombre}")
        # Salida:
        # Indice 0: Ana
        # Indice 1: Luis
        # Indice 2: Carlos
```

Código 101: Ejemplo de `enumerate`

Combinación de Técnicas Pythonicas Las técnicas pythonicas pueden combinarse para crear código más limpio y eficiente. Por ejemplo, puedes utilizar comprensiones de listas junto con funciones lambda para transformar datos de manera concisa:

```
    # Combinacion de comprension de lista con lambda
    numeros = [1, 2, 3, 4, 5]
    cuadrados = [(lambda x: x**2)(n) for n in numeros si n % 2 == 0]
    print(cuadrados)  # Salida: [4, 16]

```

Código 102: Ejemplo de código Pythonico

5.2. Arrays

En Python, los tipos de datos similares a arrays son las listas, tuplas, conjuntos y diccionarios. Cada uno tiene características y métodos específicos que facilitan la manipulación de datos. A continuación, se describen y ejemplifican cada uno de estos tipos.

5.2.1. Listas

Las listas son estructuras de datos que almacenan múltiples elementos en un solo contenedor. Son mutables, lo que permite modificar sus elementos después de la creación. Se definen con corchetes `[]` y los elementos se separan por comas. Las listas pueden contener diferentes tipos de datos, como enteros, cadenas y otros objetos.

```
    #Lista de numeros
    lista = [1, 2, 3, 4, 5]
    print(lista)

    #Lista con elementos de diferentes tipos
    mi_lista = [10, "Python", 3.14, [1, 2, 3]]
    print(mi_lista)
```

Código 103: Creación e impresión de una lista

Métodos Comunes de Listas

- **append(elemento):** Agrega un elemento al final de la lista.

```
lista = [1, 2, 3]
lista.append(4)
print(lista)
```

Código 104: Uso de `append()` para agregar un elemento al final de la lista

- **extend(iterable):** Extiende la lista agregando todos los elementos del iterable.

```
lista = [1, 2]
lista.extend([3, 4, 5])
print(lista)
```

Código 105: Uso de `extend()` para agregar múltiples elementos

- **insert(indice, elemento):** Inserta un elemento en una posición específica de la lista.

```
lista = [1, 2, 3]
lista.insert(1, "nuevo")
print(lista)
```

Código 106: Uso de `insert()` para insertar un elemento en una posición específica

- **remove(elemento):** Elimina la primera aparición del elemento de la lista.

```
lista = [1, 2, 3, 2]
lista.remove(2)
print(lista)
```

Código 107: Uso de `remove()` para eliminar un elemento

- **pop(indice):** Elimina y devuelve el elemento en la posición especificada. Si no se especifica el índice, elimina y devuelve el último elemento.

```
lista = [1, 2, 3]
elemento = lista.pop(1)
print(elemento)
print(lista)
```

Código 108: Uso de `pop()` para eliminar y devolver un elemento

- **clear():** Elimina todos los elementos de la lista.

```
lista = [1, 2, 3]
lista.clear()
print(lista)
```

Código 109: Uso de `clear()` para eliminar todos los elementos de la lista

- **index(elemento):** Devuelve el índice de la primera aparición del elemento. Lanza una excepción si el elemento no se encuentra.

```
lista = [1, 2, 3, 2]
indice = lista.index(2)
print(indice)
```

Código 110: Uso de `index()` para encontrar el índice de un elemento

- **count(elemento):** Devuelve el número de veces que el elemento aparece en la lista.

```
lista = [1, 2, 3, 2, 2]
conteo = lista.count(2)
print(conteo)
```

Código 111: Uso de `count()` para contar el número de apariciones de un elemento

- **sort(reverse=False):** Ordena los elementos de la lista en su lugar. Si se especifica `reverse=True`, se ordena en orden descendente.

```
lista = [3, 1, 2]
lista.sort()
print(lista)
```

Código 112: Uso de `sort()` para ordenar una lista

- **reverse():** Invierte el orden de los elementos en la lista.

```
lista = [1, 2, 3]
lista.reverse()
print(lista)
```

Código 113: Uso de `reverse()` para invertir el orden de los elementos

Estos son algunos métodos dentro de las listas, existen más métodos, pero esto se verá en un nivel más avanzado.

5.2.2. Tuplas

Las tuplas son estructuras de datos inmutables en Python que almacenan múltiples elementos en un solo contenedor. A diferencia de las listas, los elementos de una tupla no pueden modificarse una vez creada. Se definen con paréntesis `()` y los elementos se separan por comas. Pueden contener diferentes tipos de datos y su tamaño es fijo al momento de su creación.

```
tupla = (1, 2, 3, 4, 5)
print(tupla)

mi_tupla = (10, "Python", 3.14, (1, 2, 3))
print(mi_tupla)
```

Código 114: Creación e impresión de una tupla

Métodos Comunes de Tuplas

- **count(elemento):** Devuelve el número de veces que el elemento aparece en la tupla.

```
tupla = (1, 2, 3, 2, 2)
conteo = tupla.count(2)
print(conteo)
```

Código 115: Uso de `count()` para contar el número de apariciones de un elemento en una tupla

- **index(elemento):** Devuelve el índice de la primera aparición del elemento en la tupla. Lanza una excepción si el elemento no se encuentra.

```
tupla = (1, 2, 3, 2)
indice = tupla.index(2)
print(indice)
```

Código 116: Uso de `index()` para encontrar el índice de un elemento en una tupla

Ejemplos de Utilización de Tuplas

- Creación de una tupla y acceso a sus elementos.

```
tupla = (1, 2, 3, 4, 5)
print(tupla[0])  # Accede al primer elemento
print(tupla[1:3])  # Accede a una rebanada de la tupla
```

Código 117: Acceso a elementos y rebanadas de una tupla

- Desempaquetado de una tupla en variables individuales.

```
tupla = (10, "Python", 3.14)
a, b, c = tupla
print(a, b, c)
```

Código 118: Desempaquetado de una tupla

- Creación de una tupla anidada y acceso a elementos internos.

```
tupla = (1, (2, 3), 4)
print(tupla[1][0])  # Accede al primer elemento de la tupla anidada
```

Código 119: Acceso a elementos de una tupla anidada

- Uso de una tupla como clave en un diccionario.

```
diccionario = { (1, 2): "valor1", (3, 4): "valor2" }
print(diccionario[(1, 2)])
```

Código 120: Uso de tupla como clave en un diccionario

Como observamos los métodos que solo se pueden utilizar en tuplas, solo permiten la visualización de la tupla y de sus elementos, pero no existe ningún método para modificar una tupla, ya que es de naturaleza inmutable.

5.2.3. Conjuntos

Los conjuntos son colecciones no ordenadas de elementos únicos. Se definen con {} o `set()`, y se utilizan para operaciones de teoría de conjuntos como unión, intersección y diferencia. A diferencia de listas y tuplas, no permiten elementos duplicados ni tienen un orden específico.

```
#Ejemplo de conjunto
conjunto = {1, 2, 3, 4, 5}
print(conjunto)

#Para crear un conjunto vacio, se utiliza la funcion
conjunto_vacio = set()
print(conjunto_vacio)
```

Código 121: Creación de un conjunto

Métodos Comunes de Conjuntos Los conjuntos en Python ofrecen varios métodos útiles para manipular y consultar los elementos. A continuación se detallan algunos de los métodos más comunes:

- **add(elemento):** Agrega un elemento al conjunto. Si el elemento ya está presente, no se realiza ninguna acción.

```
conjunto = {1, 2, 3}
conjunto.add(4)
print(conjunto)
```

Código 122: Uso de `add()` para agregar un elemento a un conjunto

- **remove(elemento):** Elimina un elemento del conjunto. Lanza una excepción `KeyError` si el elemento no está presente.

```
conjunto = {1, 2, 3}
conjunto.remove(2)
print(conjunto)
```

Código 123: Uso de `remove()` para eliminar un elemento de un conjunto

- **discard(elemento):** Elimina un elemento del conjunto si está presente. Si el elemento no está presente, no hace nada.

```
conjunto = {1, 2, 3}
conjunto.discard(2)
print(conjunto)
conjunto.discard(5)  # No lanza excepcion si el elemento no esta presente
print(conjunto)
```

Código 124: Uso de `discard()` para eliminar un elemento de un conjunto

- **pop():** Elimina y devuelve un elemento arbitrario del conjunto. Lanza una excepción `KeyError` si el conjunto está vacío.

```
    conjunto = {1, 2, 3}
    elemento = conjunto.pop()
    print(elemento)
    print(conjunto)
```

Código 125: Uso de `pop()` para eliminar y devolver un elemento arbitrario

- **clear():** Elimina todos los elementos del conjunto.

```
    conjunto = {1, 2, 3}
    conjunto.clear()
    print(conjunto)
```

Código 126: Uso de `clear()` para eliminar todos los elementos del conjunto

- **union(otro_conjunto):** Devuelve un nuevo conjunto con todos los elementos que están en el conjunto original o en el conjunto pasado como argumento.

```
    conjunto1 = {1, 2, 3}
    conjunto2 = {3, 4, 5}
    union_conjunto = conjunto1.union(conjunto2)
    print(union_conjunto)
```

Código 127: Uso de `union()` para obtener la unión de dos conjuntos

- **intersection(otro_conjunto):** Devuelve un nuevo conjunto con los elementos que están en ambos conjuntos.

```
    conjunto1 = {1, 2, 3}
    conjunto2 = {3, 4, 5}
    interseccion_conjunto = conjunto1.intersection(conjunto2)
    print(interseccion_conjunto)
```

Código 128: Uso de `intersection()` para obtener la intersección de dos conjuntos

- **difference(otro_conjunto):** Devuelve un nuevo conjunto con los elementos que están en el conjunto original pero no en el conjunto pasado como argumento.

```
    conjunto1 = {1, 2, 3}
    conjunto2 = {3, 4, 5}
    diferencia_conjunto = conjunto1.difference(conjunto2)
    print(diferencia_conjunto)
```

Código 129: Uso de `difference()` para obtener la diferencia entre dos conjuntos

- **issubset(otro_conjunto):** Devuelve `True` si todos los elementos del conjunto original están en el conjunto pasado como argumento.

```
    conjunto1 = {1, 2}
    conjunto2 = {1, 2, 3}
    print(conjunto1.issubset(conjunto2))
```

Código 130: Uso de `issubset()` para verificar si un conjunto es un subconjunto de otro

- **issuperset(otro_conjunto):** Devuelve `True` si todos los elementos del conjunto pasado como argumento están en el conjunto original.

```
    conjunto1 = {1, 2, 3}
    conjunto2 = {1, 2}
    print(conjunto1.issuperset(conjunto2))
```

Código 131: Uso de `issuperset()` para verificar si un conjunto es un superconjunto de otro

Ejemplos de Utilización de Conjuntos

- Creación de un conjunto con elementos únicos y operaciones básicas.

```
    conjunto = {1, 2, 3, 4, 5}
    print(conjunto)

    # Operaciones basicas
    conjunto.add(6)
    conjunto.remove(2)
    print(conjunto)
```

Código 132: Operaciones básicas con conjuntos

- Utilización de conjuntos para eliminar duplicados de una lista.

```
    lista = [1, 2, 2, 3, 4, 4, 5]
    conjunto_sin_duplicados = set(lista)
    print(conjunto_sin_duplicados)
```

Código 133: Uso de conjuntos para eliminar duplicados

- Realización de operaciones de teoría de conjuntos como unión, intersección y diferencia.

```
    conjunto1 = {1, 2, 3}
    conjunto2 = {3, 4, 5}

    # Union
    union_conjunto = conjunto1.union(conjunto2)
    print(f"Union: {union_conjunto}")

    # Interseccion
    interseccion_conjunto = conjunto1.intersection(conjunto2)
    print(f"Interseccion: {interseccion_conjunto}")

    # Diferencia
    diferencia_conjunto = conjunto1.difference(conjunto2)
    print(f"Diferencia: {diferencia_conjunto}")
```

Código 134: Operaciones de teoría de conjuntos

Los conjuntos en Python ejemplifican la lógica de las bases de datos en cuanto a operaciones como unión, intersección y diferencia entre conjuntos de datos. Sin embargo, a diferencia de los conjuntos en Python, las bases de datos permiten la repetición de datos en las columnas de las tablas.

5.2.4. Diccionarios

Los diccionarios son colecciones de pares clave-valor donde cada clave debe ser única. Se utilizan para almacenar datos que se acceden mediante claves en lugar de índices. Se definen con llaves {} y los pares clave-valor se separan por dos puntos :. Las claves deben ser inmutables (como cadenas, números o tuplas), mientras que los valores pueden ser de cualquier tipo.

```
    diccionario = {"clave1": "valor1", "clave2": "valor2"}
    print(diccionario)
```

Código 135: Creación e impresión de un diccionario

Métodos Comunes de Diccionarios

- **get(clave, valor_por_defecto=None):** Devuelve el valor asociado a la clave especificada. Si la clave no está presente, devuelve el valor por defecto (o `None` si no se especifica).

```
    diccionario = {"clave1": "valor1", "clave2": "valor2"}
    valor = diccionario.get("clave1")
    print(valor)

    valor_inexistente = diccionario.get("clave3", "valor por defecto")
    print(valor_inexistente)
```

Código 136: Uso de `get()` para obtener valores de un diccionario

- **keys():** Devuelve una vista de las claves en el diccionario.

```
    diccionario = {"clave1": "valor1", "clave2": "valor2"}
    claves = diccionario.keys()
    print(claves)
```

Código 137: Uso de `keys()` para obtener las claves de un diccionario

- **values():** Devuelve una vista de los valores en el diccionario.

```
    diccionario = {"clave1": "valor1", "clave2": "valor2"}
    valores = diccionario.values()
    print(valores)
```

Código 138: Uso de `values()` para obtener los valores de un diccionario

- **items():** Devuelve una vista de los pares clave-valor en el diccionario.

```
    diccionario = {"clave1": "valor1", "clave2": "valor2"}
    items = diccionario.items()
    print(items)
```

Código 139: Uso de `items()` para obtener los pares clave-valor de un diccionario

- **update(diccionario_otro):** Actualiza el diccionario con los pares clave-valor del diccionario pasado como argumento. Si una clave ya existe, su valor será actualizado.

```
    diccionario = {"clave1": "valor1"}
    diccionario.update({"clave2": "valor2", "clave1": "nuevo valor"})
    print(diccionario)
```

Código 140: Uso de `update()` para actualizar un diccionario

- **pop(clave, valor_por_defecto=None):** Elimina y devuelve el valor asociado a la clave especificada. Lanza una excepción `KeyError` si la clave no está presente y no se proporciona un valor por defecto.

```
    diccionario = {"clave1": "valor1", "clave2": "valor2"}
    valor = diccionario.pop("clave1")
    print(valor)
    print(diccionario)

    # Uso de valor por defecto
    valor_inexistente = diccionario.pop("clave3", "valor por defecto")
    print(valor_inexistente)
```

Código 141: Uso de `pop()` para eliminar y devolver un valor de un diccionario

- **popitem():** Elimina y devuelve un par clave-valor arbitrario del diccionario. Lanza una excepción `KeyError` si el diccionario está vacío.

```
    diccionario = {"clave1": "valor1", "clave2": "valor2"}
    item = diccionario.popitem()
    print(item)
    print(diccionario)
```

Código 142: Uso de `popitem()` para eliminar y devolver un par clave-valor arbitrario

- **clear():** Elimina todos los pares clave-valor del diccionario.

```
    diccionario = {"clave1": "valor1", "clave2": "valor2"}
    diccionario.clear()
    print(diccionario)
```

Código 143: Uso de `clear()` para eliminar todos los pares clave-valor de un diccionario

Ejemplos de Utilización de Diccionarios

- Creación de un diccionario y acceso a sus elementos.

```
diccionario = {"nombre": "Alice", "edad": 30, "ciudad": "Lima"}
print(diccionario["nombre"])
print(diccionario["edad"])
```

Código 144: Acceso a elementos de un diccionario

- Uso de un diccionario para contar la frecuencia de palabras en una lista.

```
lista_palabras = ["manzana", "banana", "manzana", "naranja", "banana", "
manzana"]
conteo_palabras = {}

for palabra in lista_palabras:
    conteo_palabras[palabra] = conteo_palabras.get(palabra, 0) + 1

print(conteo_palabras)
```

Código 145: Uso de diccionario para contar la frecuencia de palabras

- Actualización de un diccionario con nuevos pares clave-valor.

```
diccionario = {"clave1": "valor1"}
diccionario.update({"clave2": "valor2", "clave3": "valor3"})
print(diccionario)
```

Código 146: Actualización de un diccionario con nuevos pares clave-valor

- Eliminación de un par clave-valor y manejo de claves inexistentes.

```
diccionario = {"clave1": "valor1", "clave2": "valor2"}
valor = diccionario.pop("clave2", "valor por defecto")
print(valor)
print(diccionario)
```

Código 147: Eliminación de un par clave-valor y manejo de claves inexistentes

Los diccionarios son ampliamente utilizados en la programación con Python, especialmente para la extracción y manipulación de datos. Su estructura de pares clave-valor es similar a la de los objetos en JSON (JavaScript Object Notation), un formato comúnmente utilizado para el intercambio de datos entre aplicaciones web y servidores.

Ambos, diccionarios en Python y objetos JSON, organizan datos de manera jerárquica, permitiendo almacenar información compleja de manera estructurada. En Python, los diccionarios se definen con llaves {} y separan las claves y valores con dos puntos :, mientras que JSON usa una notación similar para representar datos en formato de texto.

5.3. Slicing

En Python, el slicing (o segmentación) es una técnica que permite acceder a una subsecuencia de una secuencia, como listas, tuplas y cadenas de texto. Utiliza una notación específica para extraer partes de la secuencia, lo que facilita la manipulación de datos de manera eficiente.

```
    secuencia[inicio:fin:paso]
    #inicio: Indice donde comienza el segmento (inclusivo).
    #fin: Indice donde termina el segmento (exclusivo).
    #paso: Intervalo entre cada indice en el segmento. Es opcional y por defecto
    es 1
```

Código 148: Estructura de un slicing

```
    lista = [10, 20, 30, 40, 50]

    # Obtener los elementos del indice 1 al 3 (exclusivo)
    segmento = lista[1:4]
    print(segmento)  # Salida: [20, 30, 40]

    # Obtener todos los elementos desde el inicio hasta el indice 3
    segmento = lista[:4]
    print(segmento)  # Salida: [10, 20, 30, 40]

    # Obtener todos los elementos con un paso de 2
    segmento = lista[::2]
    print(segmento)  # Salida: [10, 30, 50]

    # Obtener los elementos en orden inverso
    segmento = lista[::-1]
    print(segmento)  # Salida: [50, 40, 30, 20, 10]
```

Código 149: Ejemplos de utilización de slicing

- **Índices Negativos:** Puedes usar índices negativos para referenciar elementos desde el final de la secuencia. Por ejemplo, secuencia[-1] se refiere al último elemento.
- **Valores por Defecto:** Si se omite inicio, se asume 0; si se omite fin, se asume hasta el final; si se omite paso, se asume 1.

5.4. Archivos

El manejo de archivos (File I/O) en Python es una parte esencial de la programación que permite leer y escribir datos en archivos.

5.4.1. Funciones de Entrada/Salida con Archivos

- **open()**: Se utiliza para abrir archivos. El modo de apertura se especifica como un segundo argumento opcional, que determina cómo se abrirá el archivo (lectura, escritura, etc.).

```
    # Abrir un archivo en modo escritura
    archivo = open("archivo.txt", "w")
    archivo.write("Hola, mundo!")
    archivo.close()
    #open() abre un archivo llamado archivo.txt en modo escritura ('w'), se
    escribe una cadena en el archivo y luego se cierra.
```

Código 150: Apertura `open()` para abrir un archivo

- **with**: Se utiliza para abrir archivos de manera segura. Garantiza que el archivo se cierre automáticamente después de que se haya terminado de usar, incluso si ocurre un error.

```
    with open("archivo.txt", "r") as archivo:
    contenido = archivo.read()
    print(contenido)
    #with abre el archivo en modo lectura ('r'), lee su contenido y lo
    imprime, y luego cierra el archivo automaticamente.
```

Código 151: Apertura segura de un archivo con `with`

- **read()**: Se utiliza para leer el contenido completo de un archivo como una cadena.

```
    with open("archivo.txt", "r") as archivo:
    contenido = archivo.read()
    print(contenido)
```

Código 152: Utilización de `read()` para leer el archivo

- **readline():** El método readline() lee una sola línea del archivo.

```
    with open("archivo.txt", "r") as archivo:
    linea = archivo.readline()
    print(linea)
```

Código 153: Utilización de `readline()` para leer una línea del archivo

- **readlines():** El método readlines() lee todas las líneas del archivo y las devuelve como una lista de cadenas.

```
    with open("archivo.txt", "r") as archivo:
    lineas = archivo.readlines()
    print(lineas)
```

Código 154: Utilización de `readlines()` para leer todas las líneas del archivo

- **write():** El método write() escribe una cadena en el archivo.

```
    with open("archivo.txt", "w") as archivo:
    archivo.write("Nueva linea de texto.")
```

Código 155: Utilización de `write()` para escribir una línea en el archivo

- **writelines():** El método writelines() escribe una lista de cadenas en el archivo.

```
    lineas = ["Primera linea.\n", "Segunda linea.\n"]
    with open("archivo.txt", "w") as archivo:
        archivo.writelines(lineas)
```

Código 156: Utilización de `writelines()` para escribir una lista de cadenas en el archivo.

5.4.2. Modos de Apertura

Los modos de apertura determinan cómo se abre un archivo y qué operaciones se pueden realizar en él:

- **a:** Abre el archivo en modo de adición append. Si el archivo no existe, se crea.

```
archivo = open("archivo.txt", "a")
archivo.write("Agrego una nueva linea.\n")
archivo.close()

# Este ejemplo abre un archivo en modo de adicion y escribe una nueva
linea en el.
```

Código 157: Apertura del archivo en modo append

- **a+:** Abre el archivo en modo de adición y lectura (append and read). Si el archivo no existe, se crea.

```
archivo = open("archivo.txt", "a+")
archivo.write("Agrego otra linea.\n")
archivo.seek(0)
contenido = archivo.read()
print(contenido)
archivo.close()

# Este ejemplo abre un archivo en modo de adicion y lectura, escribe una
nueva linea, y luego lee todo el contenido del archivo.
```

Código 158: Apertura del archivo en modo append and read

- **r:** Abre el archivo en modo de lectura (read). Si el archivo no existe, se produce un error.

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

# Este ejemplo abre un archivo en modo de lectura, lee su contenido y lo
imprime.
```

Código 159: Apertura del archivo en modo read

- **r+:** Abre el archivo en modo de lectura y escritura (read and write). Si el archivo no existe, se produce un error.

```
archivo = open("archivo.txt", "r+")
contenido = archivo.read()
print("Contenido original:", contenido)
archivo.write("Agrego contenido en modo r+.\n")
archivo.seek(0)
contenido_actualizado = archivo.read()
print("Contenido actualizado:", contenido_actualizado)
archivo.close()

# Este ejemplo abre un archivo en modo de lectura y escritura, lee su
contenido, escribe una nueva linea y luego lee el contenido actualizado.
```

Código 160: Apertura del archivo en modo `read and write`

- **"w"**: Abre el archivo en modo de escritura (write). Si el archivo no existe, se crea; si existe, se sobrescribe.

```
archivo = open("archivo.txt", "w")
archivo.write("Este es un nuevo contenido.\n")
archivo.close()

# Este ejemplo abre un archivo en modo de escritura y escribe nuevo
contenido, sobrescribiendo cualquier contenido existente.
```

Código 161: Apertura del archivo en modo `write`

- **"w+"**: Abre el archivo en modo de lectura y escritura (write and read). Si el archivo no existe, se crea; si existe, se sobrescribe.

```
archivo = open("archivo.txt", "w+")
archivo.write("Nuevo contenido en modo w+.\n")
archivo.seek(0)
contenido = archivo.read()
print(contenido)
archivo.close()

# Este ejemplo abre un archivo en modo de lectura y escritura, escribe
nuevo contenido y luego lee el contenido escrito.
```

Código 162: Apertura del archivo en modo `write and read`

6. Próximamente

6.1. Python Avanzado

- Programación Orientada a Objetos (POO)
- Manejo avanzado de archivos
- Entornos virtuales
- Instalación de paquetes
- Librerías de Python
- Crear tus propias librerías
- Control de versiones
- Generadores y funciones generadoras
- Uso avanzado de iteradores y comprensión de listas
- Expresiones regulares
- Manipulación de archivos CSV

- Manipulación de archivos JSON
- APIs y servicios web (Flask, Django)
- Manipulación de datos con pandas
- Visualización de datos con matplotlib y seaborn
- Análisis y scraping de datos web (BeautifulSoup, Scrapy)
- Manipulación de fechas y tiempos (datetime, pytz)
- Programación asíncrona (async/await)
- Multi-threading y multi-processing
- Serialización y deserialización (pickle, JSON)
- Pruebas y debugging (unittest, pytest)
- Documentación de código (docstrings, Sphinx)
- Análisis y procesamiento de datos (numpy, scipy)
- Machine Learning y Deep Learning (scikit-learn, TensorFlow, PyTorch)