Repaso de programación en Python

1. Introducción

Python, creado por Guido van Rossum y lanzado por primera vez en 1991, es uno de los lenguajes de programación más populares hoy en día. Es conocido por su simplicidad y legibilidad, lo que facilita el aprendizaje y uso tanto para principiantes como para profesionales. En este documento, exploraremos los conceptos básicos de Python, dando una mirada profunda a su sintaxis, estructuras de datos, y más, proporcionando una base sólida para aquellos interesados en adentrarse en el mundo de la programación con Python.

2. Sintaxis Básica

En esta sección, nos adentraremos en los fundamentos de la sintaxis de Python, cubriendo temas como la identación, variables, tipos de datos y operadores.

2.1 Identación

Python utiliza la identación para definir bloques de código. La identación refiere a los espacios al inicio de una línea de código. A diferencia de otros lenguajes que usan llaves para definir bloques de código, Python usa la identación, lo que lo convierte en un lenguaje visualmente más limpio y organizado.

Ejemplo:

```
if 5 > 2:
print("Cinco es mayor que dos")
```

En el ejemplo anterior, el bloque de código dentro del if está identado con dos espacios, indicando que pertenece al if.

2.1.1 Buena práctica

Es una buena práctica mantener una identación consistente en tu código, lo que generalmente se logra usando un editor de código que automáticamente aplique una identación uniforme.

2.2 Variables

Las variables son contenedores donde podemos almacenar valores. En Python, las variables se crean cuando se les asigna un valor por primera vez. Python es dinámicamente tipado, lo que significa que no necesitamos declarar el tipo de una variable al crearla.
Ejemplo:

```
x = 5
y = "Hola, mundo"
```

En el ejemplo anterior, ***x*** es una variable de tipo entero y ***y*** es una variable de tipo cadena (string).

2.2.1 Nombres de variables

Los nombres de las variables en Python pueden contener letras, números y guiones bajos, pero no pueden comenzar con un número. Es recomendable que los nombres de las variables sean descriptivos para que el código sea más legible.

2.3 Tipos de datos

Python soporta varios tipos de datos, incluyendo:

- Integers (enteros): Números sin punto decimal.
- Floats (flotantes): Números con punto decimal.
- Strings (cadenas): Secuencia de caracteres encerrados entre comillas.
- Booleans (booleanos): Valores verdadero (True) o falso (False).

Ejemplo:

```
x = 5 # Entero
y = 5.5 # Flotante
z = "Hola" # Cadena
w = True # Booleano
```

2.3.1 Casting

Python permite cambiar el tipo de una variable a través de un proceso conocido como casting. Por ejemplo, podemos cambiar un flotante a un entero usando la función int().

Ejemplo:

```
x = 5.5
x = int(x)
print(x) # Salida: 5
```

2.4 Operadores

Python ofrece una amplia gama de operadores, como operadores aritméticos para realizar operaciones matemáticas y operadores de comparación para comparar valores.

2.4.1 Operadores aritméticos

Los operadores aritm´eticos incluyen operaciones básicas como suma (+), resta (-), multiplicaci´on (*), y división (/).

Ejemplo:

```
x = 5
y = 2

print(x + y) # Suma
print(x - y) # Resta
print(x * y) # Multiplicación
print(x / y) # División
```

2.4.2 Operadores de comparaci´on

Los operadores de comparación se usan para comparar dos valores.

Incluyen: igual a (==), diferente de (!=), mayor que (>), menor que (<), mayor o igual a (>=), y menor o igual a (<=).
Ejemplo:

```
x = 5
y = 2

print(x == y) # Igual a
print(x != y) # Diferente de
print(x > y) # Mayor que
```

3. Estructuras de Datos

Las estructuras de datos son una forma de organizar y almacenar datos. Python ofrece varias estructuras de datos integradas, como listas, tuplas, diccionarios y conjuntos.

3.1 Listas

Las listas son una de las estructuras de datos más versátiles en Python, utilizadas para almacenar colecciones de ítems en una sola variable. Las listas están ordenadas, lo que significa que los ítems tienen un orden definido, y son mutables, lo que significa que podemos cambiar, agregar, y remover ítems después de que la lista esté definida.
Ejemplo:

```
mylist = ["manzana", "banana", "cherry"]
print(mylist)
```

3.1.1 Acceso a elementos

Podemos acceder a los ítems de una lista refiriéndonos al número de índice del ´ıtem. Los ´ındices en Python comienzan en 0.
Ejemplo:

```
mylist = ["manzana", "banana", "cherry"]
print(mylist[1]) # Salida: banana
```

3.1.2 Modificar elementos

Los elementos de una lista pueden modificarse refiriéndose al número de índice.
Ejemplo:

```
mylist = ["manzana", "banana", "cherry"]
mylist[1] = "arandano"
print(mylist) # Salida: [’manzana’, ’arándano’, ’cherry’]
```

3.2 Tuplas

Las tuplas son similares a las listas, pero son inmutables, lo que significa que no podemos cambiar, agregar, o remover ´ıtems una vez que la tupla está definida.
Ejemplo:

```
mytuple = ("manzana", "banana", "cherry")
print(mytuple)
```

3.2.1 Acceso a elementos

Al igual que las listas, podemos acceder a los ´ıtems de una tupla utilizando índices.
Ejemplo:

```
mytuple = ("manzana", "banana", "cherry")
print(mytuple[1]) # Salida: banana
```

3.2.2 Inmutabilidad

A pesar de que las tuplas son inmutables, es posible crear una nueva tupla con contenido modificado tomando porciones de otras tuplas.
Ejemplo:

```
mytuple = ("manzana", "banana", "cherry")
mynewtuple = mytuple[:1] + ("arandano",) + mytuple[2:] print(mynewtuple) # Salida: (’manzana’, ’arándano’, ’cherry’)
```

3.3 Diccionarios

Los diccionarios son estructuras de datos que permiten almacenar pares de clave-valor. Las claves deben ser únicas dentro de un diccionario, mientras que los valores pueden ser de cualquier tipo, y pueden repetirse.
Ejemplo:

```
mydict = {
"nombre": "Juan",
"edad": 30
}
print(mydict)
```

3.3.1 Acceso a elementos

Podemos acceder a los valores de un diccionario utilizando las claves correspondientes.
Ejemplo:

```
mydict = {
"nombre": "Juan",
"edad": 30
}
print(mydict["nombre"]) # Salida: Juan
```

3.3.2 Modificar elementos

Los valores de un diccionario pueden modificarse refiriéndose a las claves correspondientes.
Ejemplo:

```
mydict = {
"nombre": "Juan",
"edad": 30
}
mydict["edad"] = 31
print(mydict) # Salida: {'nombre': 'Juan',
'edad': 31}
```

3.4 Conjuntos

Los conjuntos son colecciones no ordenadas y sin ´ındices de ítems ´únicos. Los conjuntos son útiles para almacenar elementos sin un orden particular y para realizar operaciones de conjuntos, como unión, intersección, y diferencia.
Ejemplo:

```
myset = {"manzana", "banana", "cherry"}
print(myset)
```

3.4.1 Agregar y remover elementos

Podemos agregar y remover elementos de un conjunto usando los métodos add() y remove(), respectivamente.
Ejemplo:

```
myset = {"manzana", "banana", "cherry"}
myset.add("arandano")
myset.remove("banana")
print(myset) # Salida: {'manzana', 'cherry',
'arándano'}
```

3.4.2 Operaciones con conjuntos

Python también soporta operaciones de conjuntos tradicionales, como unión, intersección, y diferencia.
Ejemplo:

```
set1 = {"a", "b", "c"}
set2 = {"d", "e", "f", "a"}

# Union de conjuntos
print(set1.union(set2))

# Intersecci´on de conjuntos
print(set1.intersection(set2))

# Diferencia de conjuntos
print(set1.difference(set2))
```

4. Control de Flujo

El control de flujo permite dirigir el flujo de ejecución del programa mediante estructuras condicionales y bucles. En esta sección, exploraremos las estructuras condicionales (if-elif-else) y los bucles (for y while) en Python.

4.1 Condicionales (if-elif-else)

Las estructuras condicionales permiten ejecutar diferentes bloques de código dependiendo de ciertas condiciones.
Ejemplo:

```
edad = 18
if edad >= 18:
print("Mayor de edad")
else:
print("Menor de edad")
```

4.1.1 Elif

El "elif" es una abreviatura de "else if" y permite verificar múltiples expresiones para determinar si son verdaderas.
Ejemplo:

```
edad = 18
if edad > 18:
print("Mayor de edad")
elif edad == 18:
print("Justo 18 a~nos")
else:
print("Menor de edad")
```

4.2 Bucles (for, while)

Los bucles permiten ejecutar un bloque de código varias veces. Python ofrece dos tipos de bucles: for y while.

4.2.1 Bucle for

El bucle for se utiliza para iterar sobre una secuencia, que puede ser una lista, una tupla, un diccionario, un conjunto o una cadena de caracteres.

Ejemplo:

```
frutas = ["manzana", "banana", "cherry"]
for fruta in frutas:
    print(fruta)
```

4.2.2 Bucle while

El bucle while permite ejecutar un conjunto de declaraciones mientras una condición sea verdadera.

Ejemplo:

```
i = 1
while i < 6:
    print(i)
    i += 1
```

5. Funciones

Las funciones son bloques de código que están diseñados para hacer un trabajo específico. Una vez que se escribe el código para una función, se puede reutilizar en cualquier lugar de un programa. En esta sección, aprenderemos cómo definir y llamar funciones en Python, incluyendo funciones lambda.

5.1 Definición de funciones

Para definir una función en Python, usamos la palabra clave *def*, seguida del nombre de la función y una lista de parámetros entre paréntesis.

Ejemplo:

```
def saludo(nombre):
    print(f"Hola, {nombre}")
```

```
saludo("Javier")
```

5.1.1 Parámetros y argumentos

Cuando definimos una función, podemos especificar los parámetros que aceptará. Al llamar a la función, proporcionamos los valores para estos parámetros, que se conocen como argumentos.

Ejemplo:

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

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

5.2 Funciones lambda

Las funciones lambda son funciones anónimas que pueden tener cualquier número de argumentos pero solo una expresión. Se utilizan para crear fun ciones pequeñas y simples on-the-fly.

Ejemplo:

```
potencia = lambda x, y: x ** y
print(potencia(2, 3)) # Salida: 8
```

6. Módulos

Los módulos en Python son archivos que contienen código Python. Un módulo puede definir funciones, clases y variables que puedes reutilizar en otros archivos Python. En esta sección, veremos cómo importar módulos y utilizar el código que contienen.

6.1 Importación de módulos

Para utilizar un módulo en Python, primero debemos importarlo usando la instrucción import. Una vez importado, podemos acceder a las funciones y variables definidas en ese módulo.

Ejemplo:

```
import math
print(math.sqrt(16)) # Salida: 4.0
```

6.1.1 Importación selectiva

También podemos elegir importar solo algunas funciones o variables específicas de un módulo usando la sintaxis from … import …

Ejemplo:

```
from math import sqrt
print(sqrt(16)) # Salida: 4.0
```

6.1.2 Alias

Podemos darle un alias a un módulo al importarlo, lo que permite referirse a él con un nombre diferente.

Ejemplo:

```
import math as m
print(m.sqrt(16)) # Salida: 4.0
```

7. Manejo de archivos

Python facilita la lectura y escritura de archivos, permitiendo así manipular datos almacenados en disco desde un programa Python. Veamos cómo leer y escribir archivos de texto y cómo manejar errores que pueden ocurrir durante estas operaciones.

7.1 Lectura y escritura básica

Podemos leer y escribir archivos en Python usando las funciones built-in open(), read(), write(), y close().

7.1.1 Escritura a un archivo

Para escribir contenido a un archivo, usamos el método write().

Ejemplo:

```
with open('miarchivo.txt', 'w') as f:
    f.write('Hola Mundo')
```

En el código anterior, utilizamos la instrucción with para abrir el archivo en modo de escritura ('w') y escribir una cadena en él. El uso de with asegura que el archivo se cierre automáticamente al final del bloque.

7.1.2 Lectura de un archivo

Para leer el contenido de un archivo, usamos el método read().

Ejemplo:

```
with open('miarchivo.txt', 'r') as f:
        contenido = f.read()
print(contenido) # Salida: Hola Mundo
```

8 Conclusión

A través de este repaso, hemos explorado los conceptos fundamentales de Python, un lenguaje de programación versátil y poderoso. Con una comprensión sólida de estos conceptos, estás bien equipado para comenzar a desarrollar tus propios programas en Python.