Programación en Python

Introducción a Python

Como hemos aprendido, en la era digital actual, los datos son omnipresentes y fundamentales para la toma de decisiones en diversas industrias. Desde el análisis de tendencias de mercado hasta la optimización de procesos empresariales, la ciencia de datos desempeña un papel crucial en la generación de ideas y la resolución de problemas.

¿Por qué Python?

Python se ha convertido en el lenguaje de referencia en el campo de la ciencia de datos por varias razones:

Facilidad de uso: cuenta con una sintaxis clara y legible que lo hace accesible incluso para aquellos que están comenzando en programación.

Amplia variedad de bibliotecas: ofrece una amplia gama de bibliotecas especializadas en ciencia de datos, como NumPy, Pandas, Matplotlib y SciPy, que facilitan tareas como la manipulación de datos, el análisis estadístico y la visualización.

Comunidad activa: cuenta con una comunidad activa y colaborativa que constantemente desarrolla nuevas herramientas y comparte recursos educativos, lo que facilita el aprendizaje y la resolución de problemas.

Flexibilidad: es un lenguaje versátil que se puede utilizar en una variedad de contextos, desde análisis de datos hasta desarrollo web y automatización de tareas.

En esta lectura, abordaremos los conceptos fundamentales de la ciencia de datos a través de ejemplos prácticos y proyectos aplicados.

Utilizaremos Google Colab una plataforma en la nube que te permite escribir y ejecutar código Python en tu navegador, sin necesidad de instalación, para revisar cada uno de los ejemplos

¡Comencemos!

Unidad 1

Tema 1. Tipos de datos en Python

Variables

Una variable es la estructura de datos más simple que podemos encontrar. Es un espacio de memoria donde guardamos un dato, y a la vez, ese espacio de memoria recibe un nombre.

Podemos imaginarla como un contenedor o una caja donde se guardan objetos. Estos objetos pueden estar en formato texto, número, etc.

Las variables poseen un nombre llamado identificador. Es como ponerle un nombre a la caja con alguna etiqueta.
Para nombrar las variables existen ciertas reglas:

1 - El nombre no puede empezar con un número
2 - No se permite el uso de guiones medios
3 - No se permite el uso de espacios
4 - No podemos utilizar palabras reservadas

Tipos de datos en Python

En Python existen varios tipos de datos, entre los más comunes se encuentran:

- Enteros (int): números enteros positivos o negativos, sin decimales.
- Flotantes (float): números con decimales.
- Booleanos (bool): valores que representan verdadero o falso.
- Cadenas de caracteres (str): secuencias de caracteres o texto.

Operaciones entre variables

En Python, las operaciones entre variables se pueden realizar de manera similar a como se hacen con números. La forma en que se comportan estas operaciones depende del tipo de datos de las variables involucradas. Existen distintos tipos de operaciones, las aritméticas, las relacionales, las de asignación y las lógicas. Veamos algunas de ellas:

Tabla 1. Operaciones aritméticas

Operación	Descripción	Operador
Suma	Se utiliza para sumar valores numéricos o concatenar cadenas de texto.	+
Resta	Se utiliza para restar valores numéricos.	-
Multiplicación	Se utiliza para multiplicar valores numéricos o repetir cadenas de texto.	*
Potenciación	Se utiliza para elevar un número a una potencia determinada.	**
División (cociente)	Se utiliza para dividir valores numéricos.	/
División (parte entera)	Se utiliza para obtener el cociente de la división sin considerar los decimales.	//
División (resto)	Se utiliza para obtener el residuo de la división entre dos números.	%

Fuente: elaboración propia.

Tabla 2. Operaciones relacionales (comparan dos valores y devuelven un booleano true o false)

Operación	Descripción	Operador
Igual	Se utiliza para verificar si dos valores son iguales.	==
Distinto	Se utiliza para verificar si dos valores son diferentes.	!=
Mayor	Se utilizan para comparar valores numéricos o cadenas de texto y determinar si uno es mayo a otro.	<
Menor	Se utilizan para comparar valores numéricos o cadenas de texto y determinar si uno es menor que otro.	>
Mayor o igual	Se utilizan para comparar valores numéricos o cadenas de texto y determinar si uno es mayor o igual que otro.	>=
Menor o igual	Se utilizan para comparar valores numéricos o cadenas de texto y determinar si uno es menor o igual que otro.	<=

Fuente: elaboración propia.

Tabla 3. De asignación (asignan un valor a una variable)

Operación	Descripción	Operador
Simple	Se utiliza para asignar un valor a una variable.	=
Asignación con operación	Se utiliza para realizar una operación aritmética y asignar el resultado a la variable.	+ - * / // %

Fuente: elaboración propia.

Figura 1. Ejemplo

Operación	Operador
=	x=7
+=	x+=2
-=	x-=2
*=	x*=2
/=	x/=2
%=	x%=2
//=	x//=2
**=	x**=2

Fuente: elaboración propia.

Operaciones con string

Los strings son objetos inmutables con los cuales podemos utilizar varios métodos y operaciones.

Tabla 4. Operaciones con string

Operación	Descripción	Operador	Ejemplo
Concatenación	Se utiliza para asignar un valor a una variable.	+	´Hola´ + ´mundo!´ = ´Hola mundo!´
Multiplicación	Repite un *string* múltiples veces	*	"Ja" *3 = "Ja Ja Ja"

Fuente: elaboración propia.

Operaciones lógicas: and, or, xor

And se utiliza para realizar una operación de conjunción lógica entre dos expresiones booleanas. El resultado de la operación es verdadero (true), si ambas expresiones son verdaderas (true). Si una de las expresiones es falsa, el resultado será falso (false).

Figura 2. Ejemplo

A	B	A and B
1	0	0
0	0	0
0	1	0
1	1	1

Fuente: elaboración propia.

Ejemplo: la expresión "x > 0 and y < 10" será verdadera si la variable "x" es mayor que cero y la variable "y" es menor que diez. Si cualquiera de estas condiciones no se cumple, el resultado será falso.

Figura 3. Ejemplo

A	B	A or B
1	0	1
0	0	0
0	1	1
1	1	1

Fuente: elaboración propia.

Or se utiliza para realizar una operación de disyunción lógica entre dos expresiones booleanas. Si al menos una de las expresiones es verdadera (true), el resultado de la operación es verdadero (true), de lo contrario, si ambas expresiones son falsas (false), el resultado será falso (false). Por ejemplo, la expresión "x > 0 or y < 10" será verdadera si la variable "x" es mayor que cero o la variable "y" es menor que diez. Si ambas condiciones no se cumplen, el resultado será falso.

Figura 4. Ejemplo

A	B	A “or exclusiva” B
1	0	1
0	0	0
0	1	1
1	1	0

Fuente: elaboración propia.

Xor (o or exclusivo) se utiliza para realizar una operación de disyunción exclusiva lógica entre dos expresiones booleanas. Si exactamente una de las expresiones es verdadera (true), el resultado de la operación es verdadero (true), de lo contrario, si ambas expresiones son verdaderas o ambas son falsas, el resultado será falso (false).

¡Veamos los conceptos en la práctica!

Descarga el notebook haciendo clic aquí y mira el video explicativo.

Tema 2. Conversión de datos en Python (casting)

El casting en Python es el proceso de convertir un tipo de dato en otro tipo de dato. Esto se hace utilizando funciones incorporadas como "int()", "float()", "str()", "bool()", entre otras.

Por ejemplo, si queremos convertir un valor numérico almacenado como cadena en un valor entero, podemos utilizar la función "int()" para realizar la conversión. Del mismo modo, si queremos convertir un valor numérico en un valor de punto flotante, podemos utilizar la función "float()".

Números binarios y operaciones de bits

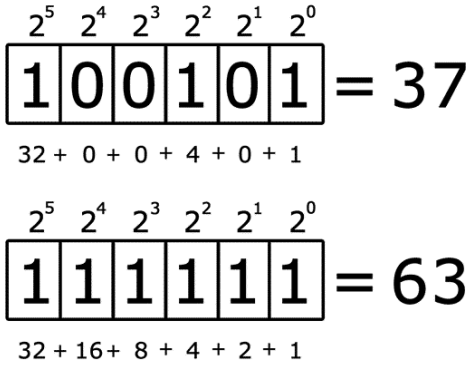
El sistema binario es un sistema numérico que utiliza únicamente dos dígitos: 0 y 1. A diferencia del sistema decimal, el que estamos más acostumbrados a usar en nuestra vida diaria, que utiliza diez dígitos (0 al 9), el sistema binario simplifica las representaciones numéricas al utilizar solo dos estados posibles.

Los números binarios y el procesamiento de datos están intrínsecamente relacionados debido a la forma en que las computadoras representan y manipulan la información. Los números binarios son la base fundamental del sistema de numeración utilizado por las computadoras.

Un bit (dígito binario) es la unidad básica de información en el sistema binario y puede tomar uno de dos valores mencionados: 0 y 1.

En un número binario, cada posición de un bit tiene un valor que es una potencia de 2. La posición más a la derecha representa 2 elevado a la potencia 0, la siguiente posición a la izquierda representa 2 elevado a la potencia 1, la siguiente 2 elevado a la potencia 2, y así sucesivamente.

Figura 5. Número binario



Fuente: elaboración propia.

Número binario: 1101

```
  1   1   0   1
2^3 2^2 2^1 2^0
```

Para convertir un número binario a su equivalente decimal, se multiplican los dígitos binarios por sus respectivas potencias de 2 y se suman los resultados. En el ejemplo anterior, el número binario 1101 se convierte a decimal de la siguiente manera:

(1 * 2^3) + (1 * 2^2) + (0 * 2^1) + (1 * 2^0) = 8 + 4 + 0 + 1 = 13

En la programación, los números binarios se utilizan en operaciones de bits para realizar manipulaciones eficientes en los datos. Estas operaciones pueden incluir desplazamientos, combinaciones lógicas (como and, or, xor) y manipulaciones más complejas utilizando las propiedades de los bits. Los números binarios son esenciales en el procesamiento de datos porque proporcionan una forma eficiente y confiable de representar, almacenar, procesar y comunicar información en sistemas digitales y computadoras.

Operaciones

Una de las operaciones de bits comunes en Python es el desplazamiento a la izquierda de bits (<<). Esta operación desplaza los bits de un número hacia la izquierda en una cantidad específica de posiciones y rellena los espacios vacíos con ceros.

Por ejemplo, consideremos la operación 1 << 2 en Python. El número binario 1 se desplaza dos posiciones hacia la izquierda y se rellena con ceros a la derecha. El resultado de esta operación es 4.

Aquí está el proceso paso a paso de la operación 1 << 2.

El número 1 en binario se representa como 0001.

Aplicamos el desplazamiento de dos posiciones hacia la izquierda a 0001, lo que resulta en 0100.

El resultado final en binario es 0100, que se traduce al número decimal 4.

```
resultado = 1 << 2
print(resultado)
```

4

La expresión resultado = 1 << 2 realiza el desplazamiento a la izquierda de bits y almacena el

resultado en la variable resultado. Luego, el valor de resultado se imprime utilizando la función print, lo que muestra el número 4 en la salida.

Flujos de control

Los flujos de control son la forma en que podemos controlar el flujo de ejecución de un programa, lo que nos permite tomar decisiones y repetir acciones en función de diferentes condiciones. Con los flujos de control, podemos hacer que un programa sea más dinámico y adaptable, y, lo mejor de esto, es que Python ofrece una gran variedad de herramientas para hacerlo de manera fácil y sencilla.

Un condicional es una estructura de control de flujo en Python que nos permite tomar decisiones y ejecutar diferentes acciones en función de si se cumple una condición o no. En otras palabras, los condicionales son una forma de programar diferentes resultados basados en ciertas condiciones

En Python, los condicionales se crean utilizando la palabra clave if seguida de una expresión booleana, seguida de un bloque de código que se ejecutará si la expresión booleana es verdadera (true). También se pueden utilizar otras palabras clave, como elif y else para ejecutar diferentes bloques de código en función de diferentes condiciones.

Ejemplo:

```
a = 5
b = 7
if a < b:
print('a es menor que b')
elif a > b:
        print('a es mayor que b')
else:
 print('a y b son iguales')
```

¡Veamos los conceptos en la práctica!

Descarga el notebook haciendo clic aquí y mira el video explicativo.

Tema 3. Ciclos operativos o loops en Python

Los ciclos operativos nos permiten repetir una acción un número determinado de veces o hasta que se cumpla una condición específica. Con los ciclos operativos, podemos automatizar tareas repetitivas y procesar grandes cantidades de datos de manera eficiente. En Python, hay dos tipos de ciclos operativos principales: for y while.

Recordemos la función de cada uno.

for

El ciclo for, en Python, se utiliza para iterar sobre una secuencia (como una lista, tupla, conjunto o cadena de texto) o cualquier objeto iterable. Itera sobre cada elemento de la secuencia y ejecuta un bloque de código para cada uno de ellos.

Su sintaxis es la siguiente:

```
for <elemento> in <elementos a iterar>:
    <Tu código>
```

Se coloca la sentencia for seguida de la variable donde se almacenarán los ítems y luego del operador in el elemento a iterar.

for e in son palabras reservadas del bucle for.

Ejemplo:

```
numeros = [1,2,3,4,5]
for num in numeros:
    suma= num+1
    print(suma)
```

while

Estructura de control de flujo que se utiliza para repetir una acción mientras se cumpla una condición específica. El bucle while ejecuta un bloque de código mientras la expresión booleana especificada en la cabecera del bucle sea verdadera (true), lo que nos permite realizar tareas repetitivas hasta que se cumpla una determinada condición.

Su sintaxis es la siguiente:

```
while expresión_booleana:
        # bloque de código a ejecutar mientras la expresión sea verdadera
```

En la primera línea, se utiliza la palabra clave while para indicar que se está creando un bucle while. Luego, se escribe una expresión booleana que se evalúa como verdadera o falsa. Si la expresión es verdadera, el bloque de código indentado que sigue se ejecutará una y otra vez hasta que la expresión sea falsa.

El bloque de código que se ejecuta dentro del bucle puede incluir cualquier cantidad de líneas de código, siempre y cuando estén indentadas (desplazadas a la derecha). Es importante tener en

cuenta que, si la expresión booleana nunca se vuelve falsa, el bucle se ejecutará infinitamente, lo que puede causar problemas en el programa. Por lo tanto, es importante asegurarse de que haya una condición en el bloque de código que permita salir del bucle eventualmente.
Ejemplo de ciclo while:

```
num = 1
while num < 6:
    print("El número es ", num)
    num =num +1
```

Sentencias break y continue

Las sentencias break y continue son dos herramientas importantes en Python que permiten controlar el flujo de ejecución de un programa.

Ambas sentencias se utilizan en bucles (como los bucles while o for) para modificar el comportamiento del programa en ciertas situaciones.

Sentencia break

La sentencia break permite alterar el comportamiento de los bucles while y for. Concretamente, nos permite terminar con la ejecución del bucle. Esto significa que una vez se encuentra la palabra break, el bucle se habrá terminado.

Ejemplos:

```
cadena = 'Python'
for letra in cadena:
    if letra == 'h':
        print("Se encontró la h")
        break
    print(letra)
```

```
x = 5
while True:
    x -= 1
    print(x)
```

```
    if x == 0:
        break
print("Fin del bucle")
```

Sentencia continue

Al igual que el ya visto break, permite modificar el comportamiento de los bucles while y for. Concretamente, la sentencia continue se salta todo el código restante en la iteración actual y vuelve al principio en el caso de que aún queden iteraciones por completar.

Ejemplo:

```
cadena = 'Python'
for letra in cadena:
    if letra == 'P':
        continue
    print(letra)
```

Veamos de forma práctica, en el siguiente video, cómo se aplican estos conceptos.

Descargar el archivo de COLAB de este módulo haciendo clic aquí.

Tema 4. Estructuras de datos en Python

Listas (list)

Las listas son estructuras de datos que pueden almacenar cualquier otro tipo de dato, inclusive una lista puede contener otra lista, además, la cantidad de elementos de una lista se puede modificar removiendo o añadiendo elementos. Para definir una lista se utilizan los corchetes, dentro de estos se colocan todos los elementos separados por comas.

```
calificaciones = [10,9,8,7.5,9]
nombres = ["Ana","Juan","Sofía","Pablo","Tania"]
```

Las listas pueden contener elementos de distintos tipos de datos, como:

```
mezcla = [True, 10.5, "abc", [0,1,1]]
```

Las listas son iterables y, por tanto, se puede acceder a sus elementos mediante indexación:

```
nombres[2]
'Sofía'
nombres[-1]
```

'Tania'

También, es posible agregar elementos a una lista mediante el método append:

```
nombres.append("Antonio")
nombres.append("Ximena")
print(nombres)
['Ana', 'Juan', 'Sofía', 'Pablo', 'Tania', 'Antonio', 'Ximena']
```

El método remove elimina un elemento de una lista:

```
nombres.remove("Ana")
print(nombres)
['Juan', 'Sofía', 'Pablo', 'Tania', 'Antonio', 'Ximena']
```

Si el valor pasado al método remove no existe, Python devolverá un ValueError:

```
nombres.remove("Jorge")
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-91-d983d2559e2f> in <module>()
----> 1 nombres.remove("Jorge")

ValueError: list.remove(x): x not in list
```

Tuplas (tuple)

Las tuplas son secuencias de elementos similares a las listas, pero se diferencian en que no pueden ser modificadas directamente. Es decir, una tupla no dispone de los métodos como append o insert que modifican los elementos de una lista.

Para definir una tupla, los elementos se separan con comas y se encierran entre paréntesis.

```
colores=("Azul","Verde","Rojo","Amarillo","Blanco","Negro","Gris")
```

Las tuplas al ser iterables pueden accederse mediante la notación de corchetes e índice. Por ejemplo:

```
colores[0]
'Azul'
colores[-1]
'Gris'
colores[3]
```

'Amarillo'

Si intentamos modificar alguno de los elementos de la tupla Python nos devolverá un TypeError:

colores[0] = "Café"

TypeError Traceback (most recent call last)

<ipython-input-96-3502c7127536> in <module>()

----> 1 colores[0] = "Café"

TypeError: 'tuple' object does not support item assignment

Diccionarios (dict)

Los diccionarios son estructuras que contienen una colección de elementos de la forma clave: valores separados por comas y encerrados entre llaves. Las claves deben ser objetos inmutables y los valores pueden ser de cualquier tipo. Necesariamente las claves deben ser únicas en cada diccionario, no así los valores.

Vamos a definir un diccionario llamado edades en el cual cada clave será un nombre y el valor una edad:

edades = {"Ana": 25, "David": 18, "Lucas": 35, "Ximena": 30, "Ale": 20}

Puede acceder a cada valor de un diccionario mediante su clave, por ejemplo, si quisiéramos obtener la edad de la clave Lucas se tendría que escribir:

edades["Lucas"]

35

Veamos estos conceptos aplicados en el siguiente video. No olvides descargar el notebook haciendo clic aquí para seguir paso a paso los ejemplos.

Tema 5. Iteradores e iterables

Los iteradores y los iterables son dos protocolos esenciales en Python, que están intrínsecamente relacionados, son ampliamente utilizados y están respaldados por una diversidad de tipos de datos.

Una de las principales razones de su importancia radica en su capacidad para integrarse fácilmente con las sentencias for en Python. Esto permite una integración sencilla de los iterables en el código, facilitando el procesamiento de secuencias de valores uno a uno.

Iterables

El protocolo iterable es utilizado por tipos donde es posible procesar su contenido uno a la vez.

Un iterable se define como un objeto que proporcionará un iterador que puede ser utilizado para recorrer y procesar sus elementos. Como tal, el iterable no es el iterador en sí mismo, sino el proveedor del iterador.

En Python, existen numerosos tipos de datos iterables, como listas, conjuntos, diccionarios, tuplas, entre otros. Todos estos son contenedores iterables que proporcionan un iterador para recorrer sus elementos.

Es posible además construir clases que generen iteradores personalizados, aunque de momento trabajaremos solo con los tipos que ya lo soportan, mencionados arriba.

Iteradores

Un iterador es un objeto que devuelve una secuencia de valores uno a uno. Pueden ser de longitud finita o de longitud infinita (aunque la mayoría de iteradores orientados a contenedores proporcionan un conjunto fijo de valores).

Por ejemplo, se puede emplear un ciclo while para recorrer los elementos de una lista y agregarlos a otra lista.

Ejemplo con un ciclo while:

```
lista = []  # Crea una lista vacía
n = -15  # Inicializa la variable n con el valor -15
while n < 0:  # Mientras n sea menor que 0, se ejecuta el bucle
    lista.append(n)  # Agrega el valor de n a la lista
    n += 1  # Incrementa el valor de n en 1 en cada iteración
print(lista)  # Imprime la lista resultante
```

Por otro lado, un ciclo for puede ser utilizado para iterar sobre los elementos de una lista y realizar operaciones basadas en ciertas condiciones.

Ejemplo con un ciclo for:

```
nombres = [‘Claudio’, ‘Marta’, ‘Horacio’]
# verificar si un valor está en la lista antes de agregarlo
elemento = 'Miguel'
if (not(elemento in nombres)):
    nombres.append(elemento)
    print('Se insertó el elemento', elemento)
```

```
else:
    print('El elemento', elemento, 'ya existía')
```

Iterar usando zip

La función zip() de Python facilita la combinación de elementos de dos listas en una tupla, permitiendo un manejo eficiente de datos en paralelo.

Viene incluida, por defecto, en el namespace, (una colección de nombres de variables y sus respectivos objetos asociados dentro de un determinado ámbito o contexto.) lo que significa que puede ser usada sin tener que importarse.

Si pasamos dos listas a zip como entrada, el resultado será una tupla donde cada elemento tendrá todos y cada uno de los elementos i-ésimos de las pasadas como entrada.

Veamos un ejemplo.

Tras aplicar zip es una lista con a[0]b[0] en el primer elemento y a[1]b[1] como segundo.

```
a = [1, 2]
b = ["Uno", "Dos"]
c = zip(a, b)

print(list(c))
# [(1, 'Uno'), (2, 'Dos')]
```

zip() con diferentes longitudes

También podemos usar zip usando iterables de diferentes longitudes. En este caso lo que pasará es que el iterador para cuando la lista más pequeña se acaba.

```
numeros = [1, 2, 3, 4, 5]
espanol = ["Uno", "Dos"]

for n, e in zip(numeros, espanol):
    print(n, e)

# 1 Uno
# 2 Dos
```

Deshacer el zip()

Con un pequeño truco, es posible deshacer el zip en una sola línea de código. Supongamos que hemos usado zip para obtener c.

```
a = [1, 2, 3]
b = ["One", "Two", "Three"]
```

```
c = zip(a, b)

print(list(c))
# [(1, 'One'), (2, 'Two'), (3, 'Three')]
```

Obtener c desde a,b:

```
c = [(1, 'One'), (2, 'Two'), (3, 'Three')]
a, b = zip(*c)

print(a)  # (1, 2, 3)
print(b)  # ('One', 'Two', 'Three')
```

*c es conocido como unpacking en Python.

Veamos ahora cómo agregar condicionales a un iterador:

```
frase = 'Todas las canciones que escuchamos'
letras_a = [i for i in frase if i == 'a']
# vamos a ir agregando a letras_a  una ‘a’ cada vez que i sea igual a ‘a’
print(len(letras_a))
# con len obtenemos la longitud de la lista letras_a
out: 4
```

Resolvemos lo anterior con un for más detallado utilizando append:

```
frase = 'Todas las canciones que escuchamos'
letras_a = []
for i in frase:
  if i == 'a':
    letras_a.append(i)
print(len(letras_a))

out: 4
```

Recuerda, para seguir paso a paso los ejemplos, descarga el notebook haciendo clic aquí.

Tema 6. Funciones

Se puede entender una función como un conjunto de líneas de código que realizan una tarea específica y pueden tomar argumentos para diferentes parámetros que modifiquen su funcionamiento y los datos de salida. Una función te permite implementar operaciones que son habitualmente utilizadas en un programa y, de esta manera, reducir la cantidad de código.

Las funciones en Python son una parte del código de nuestro programa que se encargan de cumplir algún objetivo específico definido por nosotros o por el lenguaje, recibiendo ciertos datos de entrada (argumentos) en los llamados parámetros para procesarlos y brindarnos datos de salida o de retorno.

Las funciones en Python son componentes importantes en la programación y cuentan con una estructura que consta de dos principios:

Principio de reutilización: puedes reutilizar una función varias veces y en distintos programas.

Principio de modularización: te permite segmentar programas complejos con módulos más simples para depurar y programar con mayor facilidad.

Ejemplo:

```
def di_hola(): # definimos la función di_hola
  print('Hola')

di_hola() # llamamos a la función
```

salida:

Hola

```
def bd(nombre): # definimos la función bd, pero con un parámetro (nombre)
  print('Buen día ', nombre)

bd('Juan') # invocamos a la función bd y le pasamos como argumento 'Juan'
```

salida:

Buen día Juan

Las variables dentro de las funciones tienen un alcance local, lo que significa que solo pueden ser accedidas dentro de la función donde fueron definidas. En contraste, las variables definidas fuera de las funciones tienen un alcance global y pueden ser utilizadas en cualquier parte del código. Si se necesita acceder a una variable local desde fuera de la función, se puede utilizar la palabra reservada global seguida del nombre de la variable. Es importante tener en cuenta que no se puede declarar e inicializar una variable global en una misma línea, sino que se debe hacer en

dos líneas separadas, como se muestra en el siguiente ejemplo:

```
def funcion1():
    global num1
    num1 = 10

funcion1()

print(num1)
```

El alcance de una variable determina en qué partes del código se puede acceder a ella y es crucial cuando se trabaja en proyectos grandes y complejos en los que se utilizan múltiples funciones y archivos. En estos casos, es importante tener en cuenta el alcance de una variable para evitar errores y comportamientos inesperados.

Por ejemplo, si definimos una variable global en el archivo principal de nuestro proyecto y la queremos utilizar en una función, podemos hacerlo sin problemas. Sin embargo, si definimos una variable local con el mismo nombre dentro de esta función, Python utilizará la variable local en lugar de la global. Esto puede llevar a comportamientos inesperados y errores difíciles de detectar.

```
# Definimos una variable global
nombre = "Juan"

# Creamos una función que define una variable local con el mismo nombre
def saludo():
    nombre = "Maria"
    print("Hola " + nombre)

# Llamamos a la función
saludo()

# Imprimimos la variable global
print("La variable global es " + nombre)
```

Recursividad

La recursividad o recursión es un concepto que proviene de las matemáticas, y que aplicado al mundo de la programación nos permite resolver problemas o tareas, dividiéndolas en subtareas con la misma funcionalidad. Dada la lógica de que los subproblemas a resolver son de la misma naturaleza, se puede usar la misma función para resolverlos. Dicho de otra manera, una función

recursiva es aquella que está definida en función de sí misma, por lo que se llama repetidamente a sí misma hasta llegar a un punto de salida.

Cualquier función recursiva tiene dos secciones de código claramente divididas:
por un lado, tenemos la sección en la que la función se llama a sí misma.

Por otro lado, tiene que existir siempre una condición en la que la función retorna sin volver a llamarse. Es muy importante porque de lo contrario, la función se llamaría de manera indefinida.

Calcular factorial

Uno de los ejemplos más usados para entender la recursividad, es el cálculo del factorial de un número n!.

El factorial de un número n se define como la multiplicación de todos sus números predecesores hasta llegar a uno. Por lo tanto 5!, leído como cinco factorial, sería 5*4*3*2*1.

Utilizando un enfoque tradicional no recursivo, podríamos calcular el factorial de la siguiente manera:

```
def factorial_normal(n):
    r = 1
    i = 2
    while i <= n:
        r *= i
        i += 1
    return r

factorial_normal(5)
out: 120
```

Dado que el factorial es una tarea que puede ser dividida en subtareas del mismo tipo (multiplicaciones), podemos darle un enfoque recursivo y escribir la función de la siguiente manera:

```
def factorial_recursivo(n):
    if n == 1:
        return 1
```

```
    else:
        return n * factorial_recursivo(n-1)

factorial_recursivo(5)
```

out: 120

Lo que realmente hacemos con el código anterior es llamar a la función factorial_recursivo() múltiples veces. Es decir, 5! es igual a 5 * 4! y 4! es igual a 4 * 3! y así sucesivamente hasta llegar a 1.

Algo muy importante a tener en cuenta es que si realizamos demasiadas llamadas a la función, podríamos llegar a tener un error del tipo RecursionError. Esto se debe a que todas las llamadas van apilándose y creando un contexto de ejecución, algo que podría llegar a causar un stack overflow. Es por eso por lo que Python lanza ese error, para protegernos de llegar a ese punto.

Valores por referencia y por valor

Dependiendo del tipo de dato que enviemos a la función, podemos diferenciar dos comportamientos:

paso por valor: se crea una copia local de la variable dentro de la función.

Paso por referencia: se maneja directamente la variable, los cambios realizados dentro de la función le afectarán también fuera.

Tradicionalmente:

Los tipos simples se pasan por valor: enteros, flotantes, cadenas, lógicos, etc.

Los tipos compuestos se pasan por referencia: listas, diccionarios, conjuntos, etc.

Ejemplo de paso por valor

Como ya sabemos los números se pasan por valor y crean una copia dentro de la función, por eso no les afecta externamente lo que hagamos con ellos:

```
def doblar_valor(numero):
    numero *= 2

n = 10
doblar_valor(n)
print(n)
```

10

Ejemplo de paso por referencia

Las listas u otras colecciones, al ser tipos compuestos se pasan por referencia, y si las modificamos dentro de la función estaremos modificándolas también fuera:

```
def doblar_valores(numeros):
    for i,n in enumerate(numeros):
        numeros[i] *= 2

ns = [10,50,100]
doblar_valores(ns)
print(ns)
```

```
[20, 100, 200]
```

Para modificar los tipos simples podemos devolverlos modificados y reasignarlos:

```
def doblar_valor(numero):
    return numero * 2

n = 10
n = doblar_valor(n)
print(n)
```

```
20
```

Y en el caso de los tipos compuestos, podemos evitar la modificación enviando una copia:

```
def doblar_valores(numeros):
    for i,n in enumerate(numeros):
        numeros[i] *= 2

ns = [10,50,100]
doblar_valores(ns[:])  # Una copia al vuelo de una lista con [:]
print(ns)
```

```
[10, 50, 100]
```

Funciones lambda

Es una subrutina definida que no está enlazada a un identificador. Las expresiones lambda a menudo son argumentos que se pasan a funciones de orden superior o se usan para construir el resultado de una función de orden superior que necesita devolver una función. Si la función solo

se usa una vez o un número limitado de veces, una expresión lambda puede ser sintácticamente más simple que usar una función con nombre.

En Python, las lambdas son funciones anónimas (debemos evitar asignarlas a una variable). Las funciones lambdas tienen una sintaxis concisa y pueden utilizarse con otras funciones incorporadas de gran ayuda.

Tomando de ejemplo la siguiente función:

```
def cuadrado(num):
 return num*num
```

Creamos una versión lambda de esta:

```
>>> (lambda num: num*num)(2)
```

Este es el concepto de expresión de función inmediatamente invocada, en el que llamamos a una función justo después de definirla.

Sigue practicando con el siguiente notebook haciendo clic aquí.

Tema 7. Clases y objetos (POO)

Python es un lenguaje de programación orientado a objetos. Esto quiere decir que casi todos los elementos y funcionalidades dentro del lenguaje son objetos con sus respectivas propiedades y métodos.

Pilares de la programación orientada a objetos

Abstracción: separamos los datos de un objeto y generamos un molde (una clase).

Encapsulamiento: cuando es necesario que ciertos métodos o propiedades sean inviolables o inalterables. Por ejemplo, una cuenta de banco donde el usuario no puede simplemente aumentar su balance de dinero. Depende de métodos previamente validados para poder hacerlo.

Herencia: permite crear nuevas clases a partir de otras. Por ejemplo, tenemos creada una clase “Autos” y quisiéramos crear otra clase “Auto deportivo”, podemos tomar varias propiedades y métodos de la clase “Autos”, lo que nos brinda una jerarquía padre e hijo.

Polimorfismo: proviene de poli (muchas) y morfismo (formas). Nos permite crear métodos con el mismo nombre, pero distinto comportamiento.

Clases

Las clases, por lo tanto, son plantillas o proyectos dentro de las que se guardan los objetos. Estas proporcionan un medio para agrupar datos y funcionalidades distintas.
Las clases no son objetos en Python como tal, sino que albergan a estos objetos. No obstante, la creación de una nueva clase crea un nuevo tipo de objeto, lo que permite crear nuevas instancias de ese tipo.

Objetos

Los objetos son aquellas instancias que van dentro de una clase, esto es, cualquier elemento dentro de esta. Cada una de estas instancias u objetos tiene unos atributos definidos para conservar su estado.

Asimismo, cada una de las instancias y clases pueden tener métodos dentro, definidos por su misma clase, para modificar su estado.

Así pues, una clase es como un modelo o categoría, mientras que un objeto o instancia es una copia de la clase con valores reales.

Los objetos constan de:

Estado: es el atributo o atributos de un objeto. También hace referencia a sus propiedades.

Comportamiento: son los métodos de un objeto. El comportamiento también refleja la respuesta de un objeto a otros objetos.

Identidad: es el nombre único de un objeto y permite que este interactúe con otros.

Métodos

Los métodos son funciones que están definidas dentro de una clase y operan sobre los atributos de esta, al tiempo que permiten definir las funcionalidades o responsabilidades de la clase.

El objetivo de un método dentro de las clases y objetos en Python es ejecutar las actividades que tiene encomendadas la clase a la que pertenece.
Los atributos de un objeto se modifican mediante un llamado a sus métodos.

Herencia

La herencia es un proceso mediante el cual se puede crear una clase hija que hereda de una clase padre, compartiendo sus métodos y atributos. Además de ello, una clase hija puede sobrescribir los métodos o atributos o incluso definir unos nuevos.

Se puede crear una clase hija con tan solo pasar como parámetro la clase de la que queremos heredar. En el siguiente ejemplo vemos cómo se puede usar la herencia en Python, con la clase “Perro” que hereda de “Animal”.

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

    def hacer_sonido(self):
        pass  # El método será implementado en las clases derivadas

class Perro(Animal):
    def __init__(self, nombre, edad, raza):
        # Llamamos al constructor de la clase base (Animal)
        super().__init__(nombre, edad)
        self.raza = raza

    def hacer_sonido(self):
        return "Woof!"

# Crear una instancia de la clase Perro
mi_perro = Perro(nombre="Buddy", edad=3, raza="Labrador")

# Acceder a los atributos de la clase base
print(f"{mi_perro.nombre} tiene {mi_perro.edad} años.")

# Llamar al método de la clase derivada
print(f"{mi_perro.nombre} dice: {mi_perro.hacer_sonido()}")
```

En el ejemplo anterior, “Animal” es la clase base y “Perro” es una clase derivada que hereda de “Animal”. La clase “Perro” tiene un atributo adicional (raza) y sobrescribe el método hacer_sonido

de la clase base. La función super() se utiliza para llamar al constructor de la clase base.

Dado que una clase hija hereda los atributos y métodos de la clase padre, esto nos puede ser muy útil cuando tengamos clases que se parecen entre sí, pero tienen ciertas particularidades. En este caso, en vez de definir un montón de clases para cada animal, podemos tomar los elementos comunes y crear una clase “Animal” de la que hereden el resto, respetando, por tanto, la filosofía DRY (don't repeat yourself). Realizar estas abstracciones y buscar el denominador común para definir una clase de la que hereden las demás, es una tarea de lo más compleja en el mundo de la programación.

El principio DRY es muy aplicado en el mundo de la programación y consiste en no repetir código de manera innecesaria. Cuanto más código duplicado exista, más difícil será de modificar y más fácil será crear inconsistencias.

La clase vieja se llama clase base y la que se construye a partir de ella es una clase derivada. Por ejemplo, a partir de la clase “Persona”, podemos crear la clase “Empleado” que hereda de “Persona” y le agregamos el atributo sueldo.

Sintaxis de las clases

A partir de la sentencia class y el nombre, creamos la clase. La función init() es el constructor de la clase. La clase posee atributos (especie, edad, color) y métodos que manipulan esos atributos (mePresento, cumplirAños)

```
class Persona:

    def __init__(self):
        self.nombre=input("Ingrese el nombre:")
        self.edad=int(input("Ingrese la edad:"))

    def imprimir(self):
        print("Nombre:",self.nombre)
        print("Edad:",self.edad)

class Empleado(Persona):

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

```
        self.sueldo=float(input("Ingrese el sueldo:"))

    def imprimir(self):
        super().imprimir()
        print("Sueldo:",self.sueldo)

    def paga_impuestos(self):
        if self.sueldo>3000:
            print("El empleado debe pagar impuestos")
        else:
            print("No paga impuestos")
```

Librerías en Python

Python cuenta con una amplia variedad de bibliotecas (librerías) que abarcan diversos dominios. Aquí te presento algunas de las principales y más utilizadas en distintos campos.

1. **NumPy**: biblioteca fundamental para computación numérica en Python. Proporciona soporte para arrays y matrices, así como funciones matemáticas para operar con ellas.
2. **Pandas**: herramienta poderosa para el análisis y manipulación de datos. Introduce las estructuras de datos DataFrame y Series para facilitar la manipulación de datos tabulares.
3. **Matplotlib y Seaborn**: utilizadas para la visualización de datos. Matplotlib es una biblioteca de trazado 2D y Seaborn es una interfaz de alto nivel para gráficos estadísticos.
4. **Scikit-learn**: biblioteca para aprendizaje automático y minería de datos. Proporciona herramientas simples y eficientes para análisis predictivo de datos.
5. **TensorFlow y PyTorch**: bibliotecas de aprendizaje profundo (deep learning) que permiten construir y entrenar redes neuronales.
6. **Django y Flask**: frameworks web para el desarrollo de aplicaciones web en Python. Django es un framework de alto nivel y Flask es más ligero y flexible.
7. **Requests**: biblioteca para realizar solicitudes HTTP. Es ampliamente utilizada para interactuar con servicios web.
8. **Beautiful Soup**: biblioteca para extraer información de páginas web. Se utiliza comúnmente para realizar web scraping.
9. **SQLAlchemy**: una herramienta SQL para Python que proporciona una interfaz de alto nivel para trabajar con bases de datos relacionales.
10. **Tkinter, PyQt, Kivy**: bibliotecas para la creación de interfaces gráficas de usuario (GUI). Tkinter es la biblioteca estándar, mientras que PyQt y Kivy son alternativas populares.

Estas son solo algunas de las bibliotecas más destacadas, pero la comunidad de Python es muy activa y hay muchas más bibliotecas especializadas para diferentes propósitos. Dependiendo de tus necesidades y del proyecto en el que estés trabajando, seguramente encontrarás una librería que se adapte perfectamente a tus requerimientos.

Recuerda, para seguir paso a paso los ejemplos, descarga el notebook haciendo clic aquí.

Referencias

[Imagen sin título sobre número binario]. (s. f.). https://encrypted-tbn1.gstatic.com/images?q=tbn:ANd9GcS6rf_STGy_X84p94BQgwuD8QQ1zyRO2xzy2Qc7Uu_gA4yF-UbO