Recibe una cálida:

¡Bienvenida!

Te estábamos esperando 😁

alkemy

Clase grabada

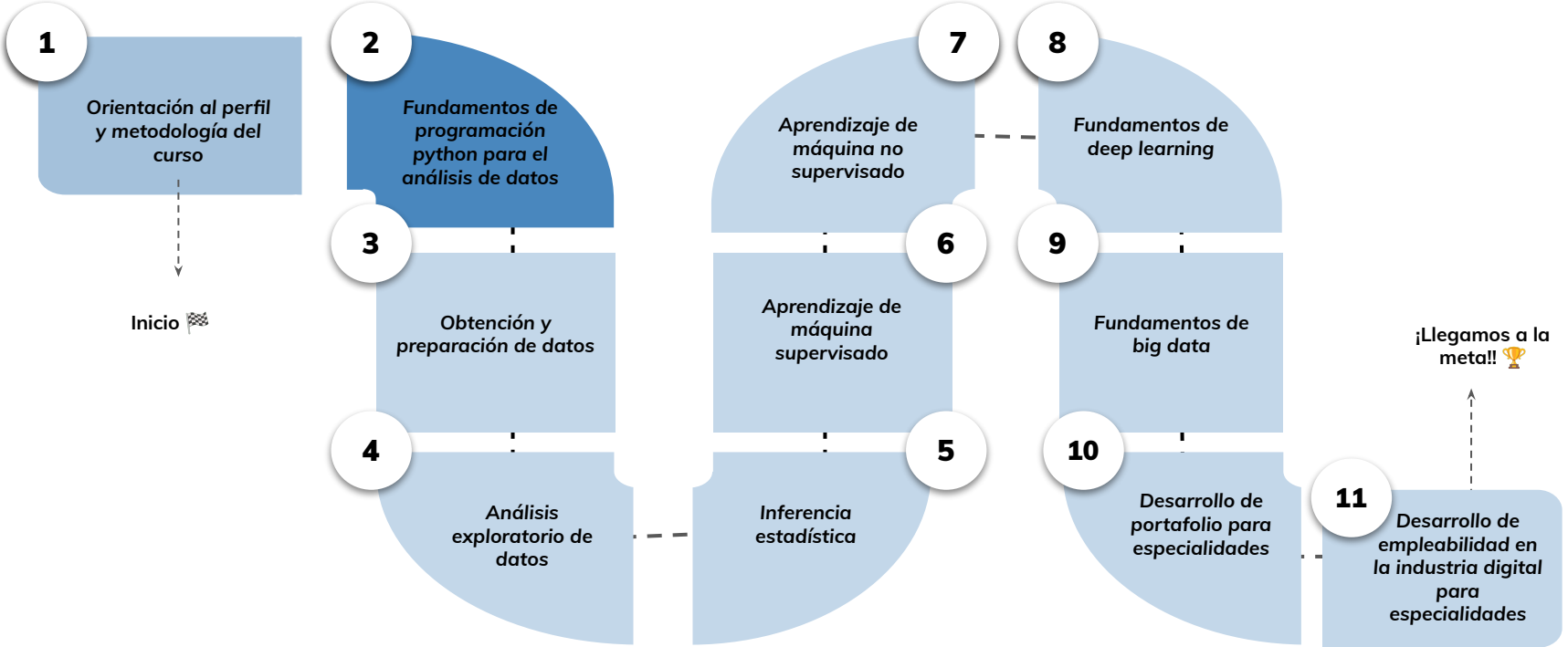
› Sentencias iterativas- Parte 2

Aprendizaje Esperado: Codificar una rutina utilizando sentencias iterativas para resolver un problema de baja complejidad en python.

alkemy

Hoja de ruta

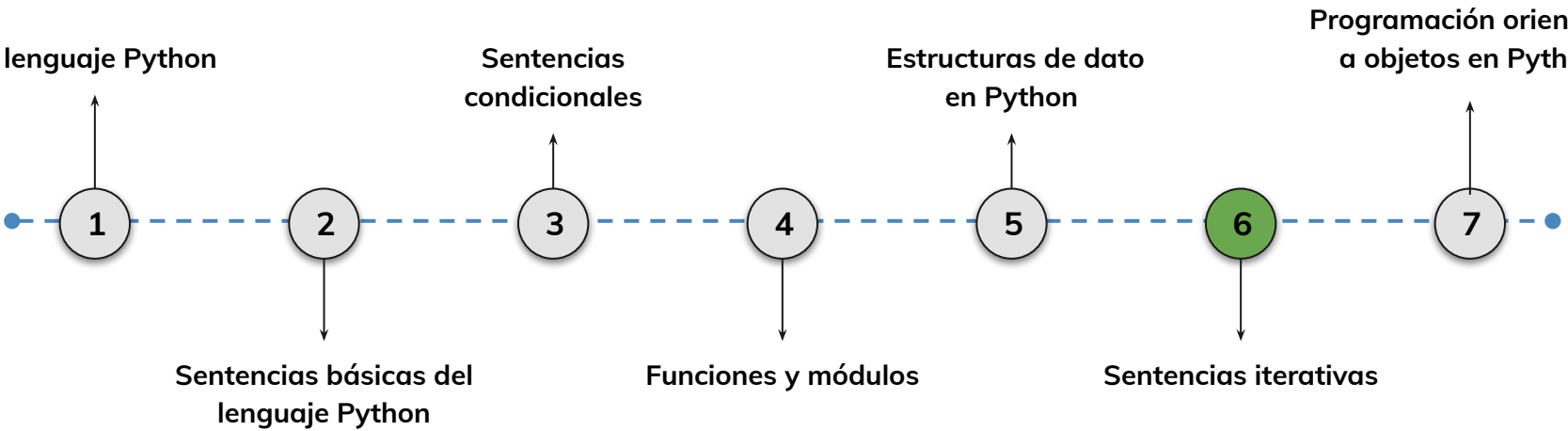
¿Cuáles skills conforman el programa? **Fundamentos de Ciencia de Datos**



alkemy

Roadmap de lecciones

*¿Cuáles **lecciones** estaremos estudiando en este módulo?*



alkemy

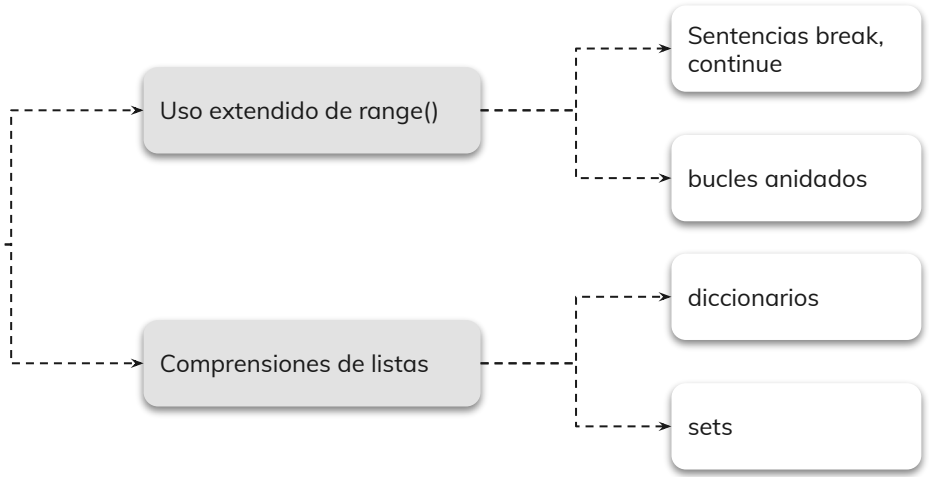
Learning Path

¿Cuáles temas trabajaremos hoy?

6.

Control del flujo de repetición en Python con herramientas avanzadas de iteración.

Estaremos profundizando en el uso de bucles for y while, incorporaremos range() en todas sus formas, veremos sentencias de control (break, continue), bucles anidados, enumerate, zip, comprensiones avanzadas e introducción a generadores.



alkemy

Objetivos de aprendizaje

¿Qué aprenderás?

- Utilizar range() con distintas configuraciones (inicio, fin, paso)
- Controlar el flujo con break, continue y bucles anidados
- Aplicar enumerate() y zip() para recorrer múltiples colecciones
- Crear comprensiones avanzadas para estructuras mutables
- Reconocer patrones comunes y optimizar el uso de bucles

Repaso clase anterior

¿Quedó alguna duda?

En la clase anterior trabajamos 📚:

- → *Aprendimos a usar for, while y range() para repetir instrucciones*
- → *Recorrimos listas y diccionarios para analizar o transformar datos*
- → *Aplicamos list comprehension para crear nuevas listas*
- → *Creamos una rutina de control de inventario usando iteración y condicionales*

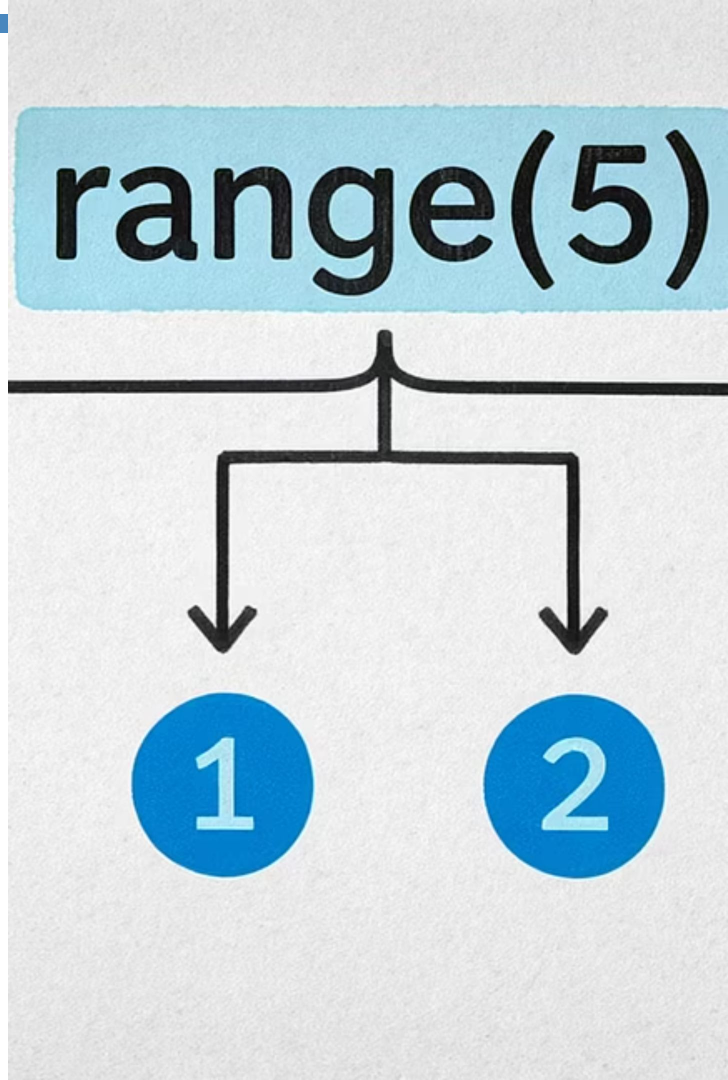
Sentencias iterativas

< >

La Función Range

La función range en Python es una herramienta muy útil para generar una secuencia de números enteros y es comúnmente utilizada en combinación con el bucle for.

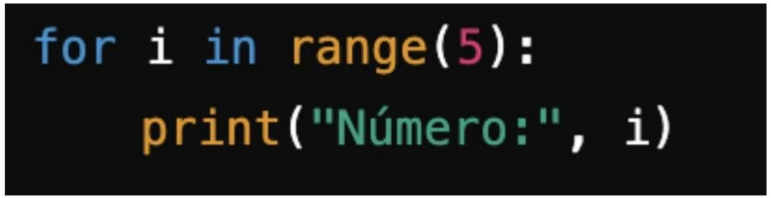
La función range tiene varias formas de uso, permitiendo definir el punto de inicio, el punto final y el incremento entre cada número de la secuencia.



Uso Básico de Range

La forma más simple de range es especificando solo el número final:

```
for i in range(5):    print(i)# Imprime: 0, 1, 2, 3, 4
```



En este caso, range(5) genera los números del 0 al 4.

Características de Range

Es importante notar que el límite superior no está incluido en la secuencia, lo cual es una característica común en Python.

Esta característica permite a range controlar de manera precisa el número de iteraciones en un bucle.

```
# range(stop) → [0, 1, 2, ..., stop-1]# range(start, stop) → [start, start+1, ..., stop-1]# range(start, stop, step) → [start, start+step, ..., stop-1]
```

< >

Range con Punto de Inicio y Fin

Range también permite especificar un punto de inicio y un punto de fin.

Por ejemplo, range(2, 6) generará una secuencia de números del 2 al 5.

Esto es útil cuando se necesita comenzar la iteración desde un número específico.

```
for i in range(2, 6):
    print("Número:", i)
```

Range con Incremento Personalizado

Además, range permite definir el intervalo entre cada número de la secuencia, lo cual es útil cuando se necesita saltar elementos o iterar en pasos mayores a uno.

Por ejemplo, range(1, 10, 2) generará números desde 1 hasta 9, en incrementos de 2.

```
for i in range(1, 10, 2):
    print("Número:", i)
```

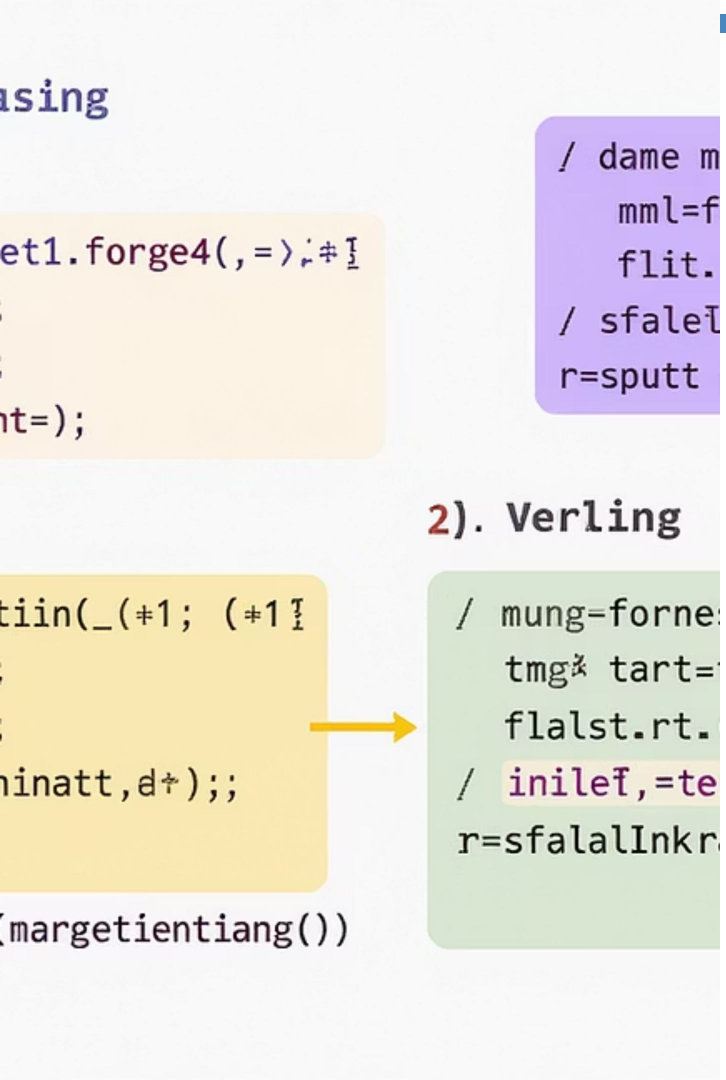
< >

Ejemplo de Range con Incremento

```
for i in range(1, 10, 2):    print(i)# Imprime: 1, 3, 5, 7, 9
```

En este ejemplo, el bucle imprimirá los números 1, 3, 5, 7 y 9.

Esto es útil en situaciones donde se necesita iterar solo sobre ciertos elementos, como al recorrer una lista en posiciones alternas o cuando se necesita un control más fino sobre las iteraciones.



Iterando con la Función Range

Combinar for con range permite crear bucles controlados en secuencias numéricas de una manera eficiente y comprensible.

Esta combinación es especialmente útil en situaciones en las que se necesita repetir una operación un número específico de veces sin preocuparse por los elementos de una lista o diccionario.

alkemy

< >

Iterando por Índice

Un caso común de uso es cuando se necesita iterar en una lista por índice. Esto permite modificar elementos específicos dentro de la lista sin necesidad de crear una copia de la misma.

```
numeros = [10, 20, 30, 40, 50]for i in
range(len(numeros)):    numeros[i] +=
5print(numeros)# [15, 25, 35, 45, 55]
```

```
numeros = [10, 20, 30, 40, 50]
for i in range(len(numeros)):
    numeros[i] += 5
print(numeros)
```

Funcionamiento de Range con Índices

En este ejemplo, range(len(numeros)) genera una secuencia de índices basada en la longitud de la lista números.

Luego, dentro del bucle, se accede y modifica cada elemento de la lista sumando 5 a cada número.

Este tipo de bucle es útil cuando se necesita modificar directamente los elementos de una lista en lugar de solo leerlos.

Repetición Controlada

Otro caso de uso común es controlar el flujo de un programa repitiendo una acción un número determinado de veces.

Esto puede ser útil en simulaciones, pruebas de rendimiento o cualquier situación en la que se necesite una repetición exacta.

```
for _ in range(3):
    print("Esta es una repetición controlada")
```

Uso de Variable Anónima

```
for _ in range(3):    print("Esta acción se repite tres veces")
```

En este caso, el guion bajo (_) se utiliza como convención para indicar que no es necesario almacenar el valor de cada iteración.

El bucle se ejecuta tres veces, imprimiendo el mismo mensaje en cada iteración.

Aplicaciones Prácticas de Range

1 Generación de Secuencias

Crear listas de números para cálculos matemáticos

2 Control de Repeticiones

Ejecutar código un número específico de veces

3 Acceso por Índice

Manipular elementos en posiciones específicas de una colección

4 Iteración Parcial

Recorrer solo ciertos elementos de una colección usando pasos

Ejemplos Avanzados con Range

```
# Iteración inversafor i in range(10, 0, -1):    print(i)  # Cuenta regresiva: 10, 9, 8, ..., 1# Acceso a elementos alternosnumeros = [10, 20, 30, 40, 50, 60]for i in range(0, len(numeros), 2):    print(numeros[i])  # Imprime: 10, 30, 50
```

Range ofrece gran flexibilidad para controlar el flujo de iteración en diferentes situaciones.

Combinando While y For

Aunque while y for tienen propósitos diferentes, a veces es útil combinarlos para resolver problemas más complejos.

```
# Buscar un elemento en una lista con límite de intentosnumeros = [10, 20, 30, 40, 50]buscar = 30intentos = 0encontrado = Falsewhile intentos < 3 and not encontrado: for num in numeros: if num == buscar: encontrado = True print(f"¡Encontrado {buscar}!") break intentos += 1
```

< >

Sentencias de Control en Bucles

1 break

Termina el bucle inmediatamente, saltándose cualquier código restante

```
for i in range(10):    if i == 5:        break
print(i)  # Imprime: 0, 1, 2, 3, 4
```

2 continue

Salta a la siguiente iteración del bucle, omitiendo el código restante en la iteración actual

```
for i in range(10): if i % 2 == 0: continue
print(i) # Imprime: 1, 3, 5, 7, 9
```

alkemy

< >

Bucles Anidados

Los bucles anidados son bucles dentro de otros bucles, útiles para trabajar con estructuras de datos multidimensionales o para realizar operaciones complejas.

```
# Imprimir una tabla de multiplicaciónfor i in range(1, 5):    for j in range(1, 5):        print(f"{i} x {j} = {i*j}")    print("-----")
```

Sin embargo, es importante tener en cuenta que los bucles anidados pueden afectar el rendimiento, ya que el número de operaciones aumenta exponencialmente.

Comprensiones Avanzadas

Python permite crear comprensiones no solo para listas, sino también para diccionarios y conjuntos, lo que facilita la transformación de datos de manera concisa.

```
# Comprensión de listacuadrados = [x**2 for x in range(10)]# Comprensión de diccionariocuadrados_dict = {x: x**2 for x in range(10)}# Comprensión de conjuntocuadrados_set = {x**2 for x in range(10)}
```

Estas técnicas son muy útiles para transformar datos de manera eficiente y con menos código.

Iteración con Enumerate

La función enumerate es útil cuando se necesita tanto el índice como el valor de cada elemento durante la iteración.

```
frutas = ["manzana", "banana", "cereza"]for indice, fruta in enumerate(frutas):    print(f"Índice {indice}: {fruta}")# Índice 0: manzana# Índice 1: banana# Índice 2: cereza
```

Esta función simplifica el código al evitar la necesidad de mantener un contador separado.

Iteración con Zip

La función zip permite iterar sobre múltiples colecciones simultáneamente, combinando elementos correspondientes en tuplas.

```
nombres = ["Ana", "Carlos", "María"]edades = [25, 30, 22]for nombre, edad in zip(nombres, edades):
print(f"{nombre} tiene {edad} años")# Ana tiene 25 años# Carlos tiene 30 años# María tiene 22 años
```

Esta función es útil para trabajar con datos relacionados almacenados en diferentes colecciones.

< >

Iteradores y Generadores

Python utiliza iteradores internamente para implementar bucles. Un iterador es un objeto que permite recorrer una colección y mantener el estado de la iteración.

Los generadores son funciones especiales que generan valores sobre la marcha, lo que permite trabajar con secuencias infinitas o muy grandes sin consumir mucha memoria.

```
# Generador simpledef contar_hasta(n): i = 1 while i
<= n: yield i i += 1# Uso del generadorfor num in
contar_hasta(5): print(num)
```

alkemy

< >

Optimización de Bucles

1 Minimizar Operaciones Costosas

Mover cálculos fuera del bucle cuando sea posible

2 Usar Funciones Incorporadas

Aprovechar funciones como map, filter y reduce para operaciones comunes

3 Evitar Bucles Anidados Innecesarios

Buscar algoritmos más eficientes para reducir la complejidad

4 Comprensiones en Lugar de Bucles

Usar comprensiones de listas, diccionarios o conjuntos cuando sea apropiado

Depuración de Bucles

La depuración de bucles puede ser desafiante debido a su naturaleza repetitiva. Algunas técnicas útiles incluyen:

- Imprimir variables clave en cada iteración
- Usar puntos de interrupción en depuradores
- Limitar el número de iteraciones durante las pruebas
- Verificar las condiciones de salida

alkemy

< >

Patrones Comunes con Bucles

1 Acumulación

Sumar o concatenar valores a lo largo de las iteraciones

```
suma = 0for num in numeros:
suma += num
```

2 Filtrado

Seleccionar elementos que cumplan ciertos criterios

```
pares = []for num in
numeros:    if num % 2 == 0:
pares.append(num)
```

3 Búsqueda

Encontrar elementos específicos en una colección

```
for elemento in lista: if
elemento == objetivo:
print("¡Encontrado!") break
```

Bucles en Programación Funcional

Python permite un enfoque funcional para operaciones que tradicionalmente se realizarían con bucles:

```
# Enfoque imperativo con buclecuadrados = []for x in range(10): cuadrados.append(x**2)# Enfoque funcionalcuadrados = list(map(lambda x: x**2, range(10)))# Filtrado imperativopares = []for x in range(10): if x % 2 == 0: pares.append(x)# Filtrado funcionalpares = list(filter(lambda x: x % 2 == 0, range(10)))
```

Iteración Asíncrona

En programación asíncrona, Python proporciona herramientas para iterar sobre colecciones sin bloquear el hilo principal:

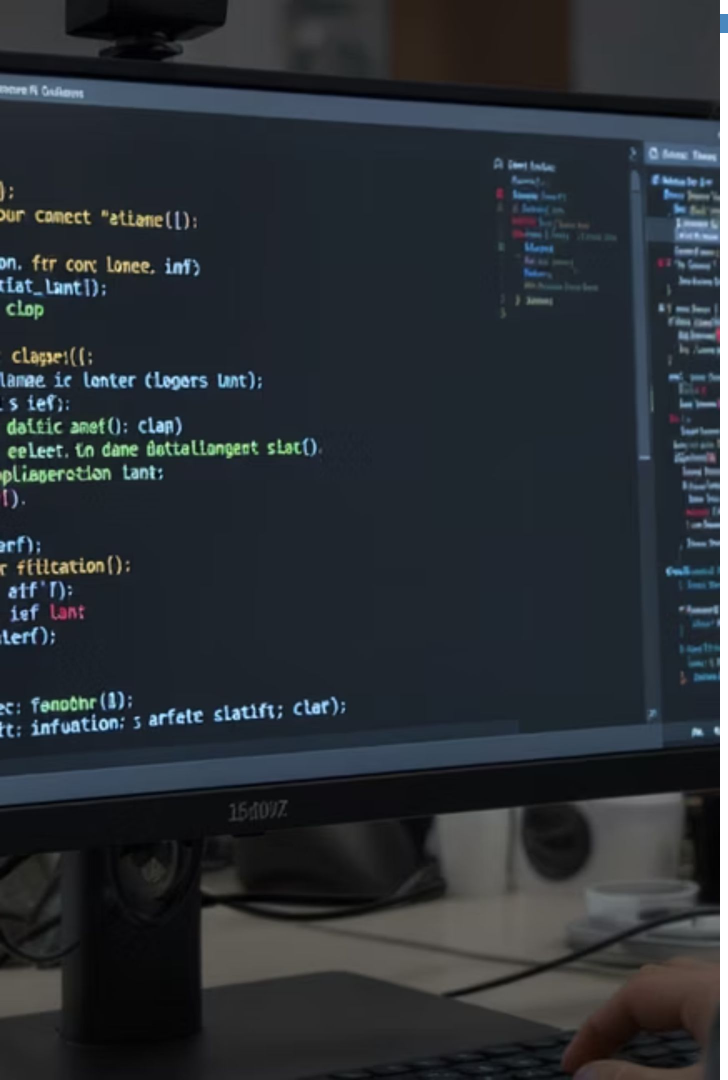
```
import asyncioasync def procesar_elemento(elemento):    await asyncio.sleep(1)  # Simula una operación asíncrona
return elemento * 2async def main():    tareas = [procesar_elemento(i) for i in range(5)]    resultados = await
asyncio.gather(*tareas)    print(resultados)asyncio.run(main())
```

Esto es útil en aplicaciones que requieren alta concurrencia, como servidores web o aplicaciones de red.

< >

Conclusiones

Las sentencias iterativas son fundamentales en la programación y permiten manejar y manipular datos de manera eficiente. Tanto while como for ofrecen formas distintas de controlar la repetición en Python, y cada una es adecuada para situaciones específicas: while es ideal cuando la repetición depende de una condición, y for es la mejor opción cuando se itera sobre una colección o secuencia con un número conocido de elementos.



Importancia de las Sentencias Iterativas

Dominar estas estructuras iterativas es esencial para cualquier programador, ya que se utilizan ampliamente en la resolución de problemas complejos y en el manejo de datos.

La capacidad de emplear correctamente while, for y range no solo mejora la eficiencia del código, sino que también permite escribir programas más organizados y fáciles de mantener.

alkemy

Live Coding

¿En qué consistirá la Demo?

Vamos a construir una rutina de control académico donde recorreremos datos de estudiantes con diferentes técnicas: bucles anidados, enumeración, zipping, comprensiones y generación de informes.

1. *Usar range(start, stop, step) para generar secuencias*
2. *Combinar listas de nombres y edades con zip()*
3. *Crear listas y diccionarios con comprensiones*
4. *Usar enumerate() para recorrer con índice*
5. *Implementar break y continue dentro de un análisis de notas*
6. *Introducir un generador con yield para listar alumnos aprobados*
7. *Mostrar cómo depurar con impresión selectiva y condiciones*

Tiempo: 25 Minutos

Momento:

Time-out!

⏰5 -10 min.

Ejercicio N° 1

Análisis académico con bucles inteligentes

Análisis académico con bucles inteligentes

Contexto: 🙌

Una escuela desea procesar la información de sus estudiantes. Para ello, necesita recorrer datos, filtrar según condiciones y generar resúmenes de forma eficiente.

Consigna: ✍️

Codificar una rutina utilizando sentencias iterativas avanzadas (for, range, enumerate, zip, comprensiones, break, continue) para analizar un listado de estudiantes y sus calificaciones.

Tiempo 🕛: *40 Minutos*

Análisis académico con bucles inteligentes

Paso a paso: ⚙️

1. Crear dos listas:

- nombres = [...]
- notas = [...] (con misma cantidad de elementos)

2. Usar zip() para recorrer ambas listas y mostrar cada nombre con su nota.

3. Usar enumerate() para identificar la posición del estudiante con la nota más baja.

4. Crear una comprensión de lista con los nombres de estudiantes aprobados (nota ≥ 6).

5. Usar for + break para detectar si hay una nota perfecta (10) y cortar el bucle al encontrarla.

6. Mostrar un resumen:

- cantidad de aprobados
- promedio de notas
- nombres en mayúsculas de estudiantes que necesitan rendir (nota < 6), usando comprensión.

¿Alguna consulta?

alkemy

Resumen

¿Qué logramos en esta clase?

- ✓ ***Utilizamos range() con múltiples parámetros***
- ✓ ***Aplicamos enumerate() y zip() para recorrer estructuras eficientemente***
- ✓ ***Implementamos comprensiones complejas de listas y diccionarios***
- ✓ ***Usamos break, continue y bucles anidados en casos reales***
- ✓ ***Introdujimos generadores para producir datos bajo demanda***
- ✓ ***Practicamos patrones comunes como acumulación y búsqueda***

alkemy

¡Ponte a prueba!

Momento de ejercitación

Te invitamos a aprovechar esta última sección del espacio sincrónico para realizar de manera individual las ***actividades disponibles en la plataforma.*** *Estas propuestas son claves para afianzar lo trabajado y* ***forman parte obligatoria del recorrido de aprendizaje.***

💪 Análisis de caso ——— 💪 Selección Múltiple

💪 Comprensión lectora

Si al resolverlas surge alguna duda, compartela o tráela al próximo encuentro sincrónico.

alkemy

¡Muchas gracias!

alkemy