**Iteradores y generadores.**

### **Iteradores.**

### **Generadores.**

**Método \_\_next\_\_**

**La expresión yield**

**Iteradores y generadores**

## **Iteradores y generadores.**

Los iteradores y generadores son conceptos fundamentales en Python que permiten trabajar con secuencias de datos de manera eficiente y en tiempo real.

Ambos se utilizan para acceder a elementos de un objeto iterable uno a la vez, pero difieren en su implementación y uso. Son objetos que cuentan con el método \_\_next\_\_ (corresponde a la función next(), el cual regresa una serie de objetos de uno en uno cada vez que es invocado.

### **Iteradores.**

Un iterador es un objeto que facilita el acceso secuencial a los elementos de una colección, como una lista o un conjunto, sin la necesidad de almacenar todos los elementos en memoria al mismo tiempo. En lugar de cargar todos los elementos en la memoria, un iterador permite obtener un elemento a la vez, lo que reduce el consumo de recursos y mejora el rendimiento.

Un iterador es un objeto que implementa los métodos \_\_iter\_\_() y \_\_next\_\_(), lo que permite recorrer secuencialmente elementos de una colección o secuencia sin cargar todos los elementos en la memoria a la vez.

El método \_\_iter\_\_() devuelve el propio objeto iterador y el método \_\_next\_\_() devuelve el siguiente elemento en la secuencia.

Cuando se llama al método \_\_next\_\_(), el iterador devuelve el siguiente elemento en la secuencia, y cuando no hay más elementos (vació), lanza la excepción StopIteration.

Los iteradores son útiles cuando necesitas recorrer secuencias de elementos uno por uno, como en un bucle for.

·

|  |  |  |
| --- | --- | --- |
| ·Código Python   |  | | --- | | nombres = ["Juan","María","Ana","Lucas","Luis", "Ariel"]  iterador\_nombres = iter(nombres)  print(f"1 {next(iterador\_nombres)=}")  print(f"2 {next(iterador\_nombres)=}")  print(f"3 {next(iterador\_nombres)=}")  print(f"4 {next(iterador\_nombres)=}")  print(f"5 {next(iterador\_nombres)=}")  print(f"6 {next(iterador\_nombres)=}")  print(f"7 {next(iterador\_nombres)=}") # comentar |   ·Salida esperada por consola   |  | | --- | | 1 next(iterador\_nombres)='Juan'  2 next(iterador\_nombres)='María'  3 next(iterador\_nombres)='Ana'  4 next(iterador\_nombres)='Lucas'  5 next(iterador\_nombres)='Luis'  6 next(iterador\_nombres)='Ariel'  Traceback (most recent call last):  File "G:\Mi unidad\Programacion\P\_cuatrimestre 2023\_2do\clase\_1.py", line 11, in <module>  print(f"7 {next(iterador\_nombres)=}")  ^^^^^^^^^^^^^^^^^^^^^^  StopIteration |   · |

·

|  |  |  |
| --- | --- | --- |
| ·Código Python   |  | | --- | | mi\_lista = [1, 2, 3, 4, 5]  iterador = iter(mi\_lista)  for elemento in iterador:  print(f"{elemento=}") |   ·Salida esperada por consola   |  | | --- | | elemento=1  elemento=2  elemento=3  elemento=4  elemento=5 |   · |

·

|  |  |  |
| --- | --- | --- |
| ·Código Python   |  | | --- | | class Contador:  def \_\_init\_\_(self, inicio, fin):  self.inicio = inicio  self.fin = fin  def \_\_iter\_\_(self):  return self  def \_\_next\_\_(self):  if self.inicio >= self.fin:  raise StopIteration  actual = self.inicio  self.inicio += 1  return actual  contador = Contador(1, 11)  iterador = iter(contador)  print(f"1 {next(iterador)=}")  print(f"2 {next(iterador)=}")  print(f"3 {next(iterador)=}")  print(f"4 {next(iterador)=}")  print(f"5 {next(iterador)=}")  for num in iterador:  print(f"next(iterador) = {num=}") # 6 a 10  exit() |   ·Salida esperada por consola   |  | | --- | | 1 next(iterador)=1  2 next(iterador)=2  3 next(iterador)=3  4 next(iterador)=4  5 next(iterador)=5  next(iterador) = num=6  next(iterador) = num=7  next(iterador) = num=8  next(iterador) = num=9  next(iterador) = num=10 |   · |

·

### **Generadores.**

Los generadores son una forma más sencilla y elegante de crear iteradores en Python. En lugar de definir una clase con métodos \_\_iter\_\_() y \_\_next\_\_(), puedes utilizar funciones generadoras.

Los generadores son objetos invocables, tales como las funciones y los métodos, los cuales implementan un método \*\\_\\_next\\_\\_()\*. Por lo general esto se realiza utilizando la expresión \_yield\_, que se utiliza para emitir valores uno a uno en lugar de return.

Un generador crea un objeto iterador automáticamente y el estado de la función se guarda entre las llamadas a yield. Esto permite pausar y reanudar la función generadora en su estado actual.

·

|  |  |  |
| --- | --- | --- |
| ·Código Python   |  | | --- | | def mi\_generador(max\_valor):  valor\_actual = 0  while valor\_actual < max\_valor:  yield valor\_actual  valor\_actual += 1  generador = mi\_generador(5)  for elemento in generador:  print(elemento) |   ·Salida esperada por consola   |  | | --- | | elemento=0  elemento=1  elemento=2  elemento=3  elemento=4 |   · |

·

Los generadores son especialmente útiles cuando trabajas con grandes conjuntos de datos o cuando no necesitas mantener todos los elementos en memoria al mismo tiempo.

En resumen, tanto los iteradores como los generadores son técnicas esenciales para trabajar con secuencias de datos de manera eficiente y elegante en Python.

·

|  |  |  |
| --- | --- | --- |
| ·Código Python   |  | | --- | | def generador\_pares(maximo):  num = 0  while num < maximo:  yield num  num += 2  gen = generador\_pares(11)  for numero in gen:  print(f"{numero=}") |   ·Salida esperada por consola   |  | | --- | | numero=0  numero=2  numero=4  numero=6  numero=8  numero=10 |   · |

**Método \_\_next\_\_:**

El método \_\_next\_\_ es un método especial que permite implementar la funcionalidad de un iterador en una clase personalizada. Cuando se llama a next() en un objeto que implementa este método, se obtiene el siguiente elemento en la secuencia que el iterador representa. Si no hay más elementos, se levanta la excepción StopIteration para indicar que la iteración ha finalizado.

La palabra clave yield se utiliza en funciones para crear generadores en Python. Un generador es una función especial que produce una secuencia de valores bajo demanda y permite pausar y reanudar la ejecución de la función sin perder su estado interno. Cuando se encuentra una declaración yield en una función, esta se pausa y devuelve el valor indicado. En la siguiente llamada a la función, la ejecución se reanuda desde donde se detuvo, manteniendo el estado.

**La expresión yield .**

Esta expresión regresa un objeto del ámbito local de una función al ámbito superior de ésta, pero a diferencia de \_return\_ la función no es terminada, sino que continúa su ejecución.

|  |  |  |
| --- | --- | --- |
| ·Código Python   |  | | --- | | def numeros\_pares(maximo):  num = 0  while num < maximo:  yield num  num += 2  generador = numeros\_pares(11)  print(f"1 {next(generador)=}")  print(f"2 {next(generador)=}")  print(f"3 {next(generador)=}")  print(f"4 {next(generador)=}")  print(f"5 {next(generador)=}")  print(f"6 {next(generador)=}")  print(f"7 {next(generador)=}") # comentar |   ·Salida esperada por consola   |  | | --- | | 1 next(generador)=0  2 next(generador)=2  3 next(generador)=4  4 next(generador)=6  5 next(generador)=8  6 next(generador)=10  Traceback (most recent call last):  File "G:\Mi unidad\Programacion\P\_cuatrimestre 2023\_2do\clase\_1.py", line 16, in <module>  print(f"7 {next(generador)=}")  ^^^^^^^^^^^^^^^  StopIteration |   · |

·

|  |  |  |
| --- | --- | --- |
| ·Código Python   |  | | --- | | def generador\_pares(maximo):  num = 0  while num < maximo:  yield num  num += 2  gen = generador\_pares(11)  for numero in gen:  print(f"{numero=}") |   ·Salida esperada por consola   |  | | --- | | numero=0  numero=2  numero=4  numero=6  numero=8  numero=10 |   · |

·

En este ejemplo, la función numeros\_pares es un generador que produce números pares hasta un valor máximo. La declaración yield num pausa la función y devuelve el valor de num en cada iteración. La próxima vez que se llame a la función, continuará desde donde se detuvo y seguirá generando valores.

|  |  |  |
| --- | --- | --- |
| ·Código Python   |  | | --- | | class MiIterador:  def \_\_init\_\_(self, max\_valor):  self.max\_valor = max\_valor  self.valor\_actual = 0  def \_\_iter\_\_(self):  return self  def \_\_next\_\_(self):  if self.valor\_actual < self.max\_valor:  self.valor\_actual += 1  return self.valor\_actual - 1  else:  raise StopIteration  iterador = MiIterador(5)  for elemento in iterador:  print(f"{elemento=}") |   ·Salida esperada por consola   |  | | --- | | elemento=0  elemento=1  elemento=2  elemento=3  elemento=4 |   · |

·

**Lazy evaluation:**

La evaluación perezosa es una estrategia de evaluación utilizada en algunos lenguajes de programación que consiste en posponer la evaluación de una expresión hasta que sea necesaria o se requiera su resultado. En lugar de calcular todos los valores de una expresión de forma inmediata, la evaluación perezosa permite evaluar solo los valores necesarios en un momento dado.

La evaluación perezosa es especialmente útil cuando se trabaja con estructuras de datos potencialmente grandes o cuando se tienen expresiones complejas donde no todos los valores son necesarios en todos los casos. Al posponer la evaluación, se puede evitar el cálculo innecesario de valores que no se utilizarán.

En Python, la evaluación perezosa se puede lograr utilizando generadores, que son objetos que generan una secuencia de valores bajo demanda. Los generadores se definen utilizando la sintaxis de comprensión de listas pero con paréntesis en lugar de corchetes.

·

|  |  |  |
| --- | --- | --- |
| ·Código Python   |  | | --- | | todos = [1, 2, 3, 4, 5, 6, 7, 8, 9, 0]  cuadrados\_en\_reserva = (x \*\* 2 for x in todos)  # los primeros 3 cuadrados  print(f"primer valor:",next(cuadrados\_en\_reserva)) # 1  print(f"segundo valor:",next(cuadrados\_en\_reserva)) # 4  print(f"tercer valor:",next(cuadrados\_en\_reserva)) # 9 |   ·Salida esperada por consola   |  | | --- | | primer valor: 1  segundo valor: 4  tercer valor: 9 |   · |

·

En este ejemplo, en lugar de generar una lista completa con los cuadrados de los números, se crea un generador cuadrados\_en\_reserva que produce los cuadrados bajo demanda. Al llamar a la función next(cuadrados\_en\_reserva ), se obtiene el siguiente valor del generador.

La evaluación perezosa tiene la ventaja de ahorrar recursos y tiempo de ejecución al evitar cálculos innecesarios. Sin embargo, es importante tener en cuenta que los generadores son un tipo de iteradores y solo pueden ser recorridos una vez. Una vez que se agotan los valores del generador, no se pueden obtener más elementos.

En resumen, es una estrategia que permite posponer la evaluación de expresiones hasta que sea necesario, lo que puede ser beneficioso en términos de eficiencia y rendimiento en ciertos escenarios