Seminario 16:

Iteradores y generadores en Python.

En diseño de software, el patrón de diseño **Iterador** , define una interfaz que declara los métodos necesarios para acceder secuencialmente a un grupo de objetos de una colección.

Este patrón de diseño permite recorrer una estructura de datos sin que sea necesario conocer la estructura interna de la misma.

Para la creación del patrón iterador debe implementarse el control de la iteración (pudiendo ser un iterador externo que ofrece los métodos para que el cliente recorra la estructura paso a paso, o un iterador interno que ofrece un método de actuación sobre la estructura que, de manera transparente al cliente, la recorre aplicándose a todos sus elementos) y definirse el recorrido. A mayores se podrían implementar operaciones adicionales en el iterador o definir la estructura de éste de una manera más robusta ante cambios en la estructura. Hay que tener especial cuidado en la implementación de iteradores con accesos privilegiados, iteradores para estructuras compuestas o iteradores nulos.

El código basado en iteradores ofrece mejores características de mejor consumo de memoria que el código que usa listas. Dado que los datos no se producen a partir del iterador hasta que sea necesario, no es necesario que todos los datos sean almacenados en la memoria al mismo tiempo. Este modelo de procesamiento «perezoso» puede reducir el intercambio y otros efectos secundarios de grandes conjuntos de datos, mejorando el rendimiento.

En Python hay diversas funciones que se encargan del trabajo con iteradores (vamos a mencionar algunas solamente en este documento).

La función *chain()* toma varios iteradores como argumentos y devuelve un único iterador que produce el contenido de todas las entradas como si vinieran de un solo iterador, *chain()* facilita el procesamiento de varias secuencias sin construir una lista grande.

```
from itertools import *

for i in chain([1, 2, 3], ['a', 'b', 'c']):
    print(i, end=' ')
print()
```

```
1 2 3 a b c
```

La función incorporada *zip()* devuelve un iterador que combina los elementos de varios iteradores en tuplas. Al igual que con las otras funciones de este módulo, el valor de retorno es un objeto iterable que produce valores uno a la vez. *zip()* se detiene cuando el primer iterador de entrada se agota. Para procesar todas las entradas, incluso si los iteradores producen diferentes números de valores, usa *zip_longest()*.

```
for i in zip([1, 2, 3], ['a', 'b', 'c']):
    print(i)

(1, 'a')
(2, 'b')
(3, 'c')
```

La función *iter()* se suele emplear para mostrar cómo funciona en realidad un bucle implementado con *for/in*. Antes del inicio del bucle la función *iter()* retorna el objeto iterable con el método subyacente *__iter__()*. Una vez iniciado el bucle, el método *__next__()* permite avanzar, en cada ciclo, al siguiente elemento hasta alcanzar el último. Cuando el puntero se encuentra en el último elemento si se ejecuta nuevamente el método *__next__()* el programa produce la excepción **StopIteration**:

```
lista = [10, 100, 1000, 10000]
iterador = iter(lista)
try:
    while True:
            print(iterador.__next__())
except StopIteration:
    print("Se ha alcanzado el final de la lista")
```

La diferencia entre un iterable y un iterador es algo sutil. Un iterador es un objeto que se puede obtener de un iterable a través de la función iter. En realidad, podemos implementar un bucle tanto con un iterable como con un iterador.

Básicamente, un iterable es un objeto capaz de devolver sus miembros de uno en uno.

```
a = [1, 2, 3]
b = iter(a)
for x in b:
    print(x, end=' ')
```

```
1 2 3
```

Los **generadores** son una forma sencilla y potente de iterador. Un generador es una función especial que produce secuencias completas de resultados en lugar de ofrecer un único valor. En apariencia es como una función típica pero en lugar de devolver los valores con *return* lo hace con la declaración *yield*. Hay que precisar que el término generador define tanto a la propia función como al resultado que produce.

Una característica importante de los generadores es que tanto las variables locales como el punto de inicio de la ejecución se guardan automáticamente entre las llamadas sucesivas que se hagan al generador, es decir, a diferencia de una función común, una nueva llamada a un generador no inicia la ejecución al principio de la función, sino que la reanuda inmediatamente después del punto donde se encuentre la última declaración *yield* (que es donde terminó la función en la última llamada).

```
# Declara generador
def gen_basico():
    yield "uno"
    yield "dos"
    yield "tres"

for valor in gen_basico():
    print(valor)                    # uno, dos, tres

# Crea objeto generador y muestra tipo de objeto
generador = gen_basico()
print(generador)                # generator object gen_basico
print(type(generador))          # class 'generator'

# Convierte a lista el objeto generador y muestra elementos
lista = list(generador)
print(lista)                    # ['uno', 'dos', 'tres']
print(type(lista))              # class 'list'
```

El siguiente generador produce una sucesión de 10 valores numéricos a partir de un valor inicial. El valor final se obtiene sumando 10 al inicial y el bucle se ejecuta mientras el valor inicial es menor que el final. El ejemplo muestra como se almacenan los valores de las variables en cada ciclo y el punto donde se reanuda el bucle en cada llamada.

```
def gen_diez_numeros(inicio):
    fin = inicio + 10
    while inicio < fin:
        inicio+=1
        yield inicio, fin
for inicio, fin in gen_diez_numeros(23):
    print(inicio, fin)
```

En un generador la declaración *yield* puede aparecer en varías líneas e incluso dentro de un bucle. El intérprete Python producirá una excepción de tipo **StopIteration** si encuentra el comando *return* durante la ejecución de un generador.

Conjunto de Wirth (Iterador):

```
class WirthIterableIterator:
    def __init__(self):
        self.cola = [1]
    def __iter__(self):
        return self
    def __next__(self):
        value = self.cola.pop()
        self.cola.insert(0, value*2+1)
        self.cola.insert(0, value*3+1)
        return value
```

Conjunto de Wirth (Generador):

```
def WirthIterableGenerator():
    cola = [1]
    while True:
        value = cola.pop()
        cola.insert(0, value*2+1)
        cola.insert(0, value*3+1)
        yield value
```

ConcatIterable(Iterador):

```
class ConcatIterableIterator:
    def __init__(self, *args):
        self.Iters = [it for it in args]
        self.Ind_iter_act = 0
        self.UpdateIter()

    def UpdateIter(self):
        self.Iter_act = iter(self.Iters[self.Ind_iter_act])

    def __iter__(self):
        return self
    def __next__(self):
        try:
            value = next(self.Iter_act)
            return value
        except StopIteration:
            if self.Ind_iter_act == len(self.Iters)-1:
                self.Ind_iter_act = 0
                self.UpdateIter()
                raise StopIteration
            self.Ind_iter_act+=1
            self.UpdateIter()
            value = next(self.Iter_act)
            return value
```

ConcatIterable (Generador):

```
def ConcatIterableGenerator(*args):
    for It in args:
        for i in It:
            yield i
```

WhereIterable (Iterable):

```
class WhereIterableIterator:
    def __init__(self, It, pred):
        self.It = iter(It)
        self.pred = pred
    def __iter__(self):
        return self
    def __next__(self):
        value = next(self.It)
        while not self.pred(value):
            value = next(self.It)
        return value
```

WhereIterable (Generador):

```
def WhereIterableGenerator(It, pred):
    It = iter(It)
    for i in It:
        if pred(i):
            yield i
```

Expresiones Generadoras y List Comprehension:

Estos conceptos son bastante parecidos en su sintaxis, la única diferencia es que el primero se escribe entre paréntesis y el segundo entre corchetes.

Las expresiones generadoras devuelven un objeto generador que produce resultados por demanda.

```
Gen = (x**2 for x in range(1000))
```

En cada iteración del objeto Gen nos va a devolver el valor que devuelva el iterador range(1000) aplicándole la función x**2.

Fijense que también podríamos haber implementado el WhereIterable con generador de la siguiente forma:

```
where = (x for x in iterable if pred(x))
```

List Comprehension:

Las List Comprehensions coleccionan el resultado de aplicar una expresión a una secuencia de valores y los retornan en una nueva lista.

Supongamos que queremos guardar el código ASCII de todos los caracteres de un string. Quizas lo primero que se nos ocurra sea algo como esto:

```
res = []
for x in 'listComp':
    res.append(ord(x))
```

Pero utilizando una expresión de List Comprehension podemos lograr el mismo resultado con algo así:

```
res=[ord(x) for x in 'listComp']
```