MODUL 3

Iterator, Generator, dan Variasi Fungsi

Praktikum Pemrograman Berbasis Fungsi

SAINS DATA
INSTITUT TEKNOLOGI SUMATERA

3.1 Tujuan

1. Memahami konsep iterator dan generator dalam Python.
2. Mampu mengimplementasikan iterator dan generator untuk berbagai kebutuhan pemrograman.
3. Memahami perbedaan antara iterator dan generator.

3.2 Konsep Dasar

Memahami iterasi, iterator, dan generator penting untuk menulis kode yang efisien dan hemat memori, terutama saat menangani data besar. Iterator mengizinkan kita mengulang urutan data satu elemen pada satu waktu. Iterator bekerja dengan menggunakan metode penting, yakni *iter* dan *next* untuk mengontrol perulangan. Sementara generators adalah fungsi khusus yang menghasilkan nilai secara bertahap menggunakan *yield*. Generator bekerja dengan menghasilkan nilai secara lengkap saat diperlukan, menghindari alokasi memori besar.

3.2.1 Iterasi dan Iterable

Iterasi merupakan proses mengakses elemen dalam suatu kumpulan data secara berurutan, satu per satu, dengan menggunakan struktur perulangan. Iterable adalah objek yang dapat diiterasi, seperti list, tuple, string, dan jenis data kolektif lainnya. Secara sederhana, setiap objek yang dapat digunakan dalam pernyataan for in dapat dikategorikan sebagai iterable.

3.2.2 Iterator

Iterator berbeda dari iterable. Iterator adalah objek yang dapat diiterasi dan memiliki kemampuan untuk mengingat status perulangannya. Dengan demikian, iterator dapat melacak posisi saat ini dalam iterasi, menentukan elemen berikutnya, serta mengetahui kapan iterasi harus dihentikan.

Sebuah iterable dapat dikonversi menjadi iterator dengan menggunakan fungsi iter().Selanjutnya, kita mengenal fungsi next(), yang digunakan untuk mengambil elemen berikutnya dalam sebuah iterator. Fungsi ini memungkinkan kita untuk mengakses data iterasi satu per satu secara manual. Berikut adalah contoh penggunaannya:

```
>>> x = iter([1, 2, 3])
>>> x
<listiterator object at 0x1004ca850>
>>> next(x)
1
>>> next(x)
2
>>> next(x)
3
>>> next(x)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
StopIteration
```

Penggunaan fungsi next() terhadap object iterator yang sudah terakses semua itemnya menghasilkan error. Manfaatkan keyword try except jika diperlukan.

Iterator memungkinkan kita untuk menelusuri elemen satu per satu dalam suatu koleksi data tanpa harus mengakses semua elemen sekaligus.
Objek dikatakan sebagai iterator jika memiliki dua metode:
1) __iter__() : Mengembalikan objek iterator itu sendiri.
2) __next__() : Mengembalikan elemen berikutnya dalam urutan.

Berikut contoh fungsi iterator untuk menampilkan angka berurutan.

Fungsi iterator

```
class myrange:
    def __init__(self, n):
        self.i = 0
        self.n = n

    def __iter__(self):
        return self

    def __next__(self):
        if self.i < self.n:
            i = self.i
            self.i += 1
            return i
        else:
            raise StopIteration()
```

Hasil

```
>>> my_r = myrange(3)
>>> next(my_r)
0
>>> next(my_r)
1
>>> next(my_r)
2
>>> next(my_r)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 14, in __next__
StopIteration
```

Penjelasan:

- Metode __iter__ memungkinkan suatu objek untuk dapat diiterasi.
- Secara internal, pemanggilan fungsi iter() akan menjalankan metode __iter__ pada objek yang diberikan.
- Metode __iter__ harus mengembalikan sebuah iterator,
- Iterator tersebut harus memiliki metode __next__ dan harus mengeluarkan StopIteration ketika tidak ada elemen lagi.

3.2.3 Generator

Jika kita ingin membuat sebuah iterator, kita dapat melakukannya seperti yang telah kita lihat sebelumnya, atau kita dapat menggunakan metode yang lebih sederhana, yang disebut generator. Generator menggunakan kata kunci `*yield*` untuk menghasilkan nilai satu per satu tanpa harus menyimpan semua elemen di memori.

Perbedaan utama antara iterator dan generator:

Aspek	Iterator	Generator
Cara kerja	Menggunakan `__iter__()` dan `__next__()`	Menggunakan `yield`
Penyimpanan	Menyimpan seluruh elemen dalam memori	Hanya menyimpan nilai terakhir
Efisiensi	Bisa menghabiskan banyak memori	Lebih efisien dalam penggunaan memori

Berikut adalah fungsi generator yang menghasilkan sebuah generator. Dimana setiap kali pernyataan yield dieksekusi, fungsi akan menghasilkan nilai baru.

Fungsi generator

```
def myrange(n):
    i = 0
    while i < n:
        yield i
        i += 1
```

Hasil

```
>>> my_r = myrange(3)
>>> my_r
<generator object yrange at 0x401f30>
>>> next(my_r)
0
>>> next(my_r)
1
>>> next(my_r)
2
>>> next(my_r)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
StopIteration
```

Bagaimana fungsi generator bekerja secara internal?, Berikut adalah fungsi generator yang telah dimodifikasi dan ditambahkan beberapa pernyataan print.

Fungsi generator

```
>>> def foo():
...     print("begin")
...     for i in range(3):
...         print("before yield", i)
...         yield i
...         print("after yield", i)
...     print("end")
...
```

Hasil

```
>>> f = foo()
>>> next(f)
begin
before yield 0
0
>>> next(f)
after yield 0
before yield 1
1
>>> next(f)
after yield 1
before yield 2
2
>>> next(f)
after yield 2
end
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
StopIteration
>>>
```

Ketika sebuah fungsi generator dipanggil, ia mengembalikan sebuah objek generator tanpa langsung mengeksekusi fungsi tersebut. Saat metode next dipanggil untuk pertama kali, fungsi mulai dieksekusi hingga mencapai pernyataan yield. Nilai yang dihasilkan akan dikembalikan oleh pemanggilan next tersebut.

Kita dapat membuat sebuah generator menggunakan "**generator expression**", yang mirip dengan ekspresi list yang biasa kita gunakan. Berikut sebagai contoh.

```
>>> a = (x*x for x in range(10))
>>> a
<generator object <genexpr> at 0x401f08>
>>> sum(a)
285
```

Perhatikan bahwa kita telah mengganti tanda kurung siku pada list comprehension dengan tanda kurung biasa — ini akan membuat sebuah generator.

Ingat: generator tidak menghasilkan semua nilainya sekaligus, sehingga dapat meningkatkan kinerja.

Glossary

Istilah	Definisi
Iterable	Objek Python yang dapat diulang atau diiterasi dalam sebuah loop. Contoh iterable termasuk list, set, tuple, dictionary, string, dll.
Iterator	Objek yang dapat diiterasi. Iterator berisi sejumlah nilai yang dapat dihitung.
Generator	Jenis fungsi khusus yang tidak mengembalikan satu nilai saja, tetapi mengembalikan objek iterator dengan serangkaian nilai.
Lazy Evaluation	Strategi evaluasi di mana objek hanya dibuat saat dibutuhkan. Oleh karena itu, dalam beberapa komunitas pengembang, lazy evaluation juga disebut sebagai "call-by-need".
Iterator Protocol	Seperangkat aturan yang harus diikuti untuk mendefinisikan sebuah iterator dalam Python.
next()	Fungsi bawaan yang digunakan untuk mengembalikan item berikutnya dalam sebuah iterator.
iter()	Fungsi bawaan yang digunakan untuk mengubah iterable menjadi iterator.
yield	Kata kunci dalam Python yang mirip dengan return, tetapi yield mengembalikan objek generator, bukan nilai langsung.

3.3 Prosedur Percobaan

3.3.1 Studi kasus 1 : Membuat fungsi iterator dan generator yang menghasilkan pangkat tiga angka dari 1 hingga n. Inputkan n dari pengguna.

Penjelasan:
Dapat kita lihat, Iterator dibuat dengan mendefinisikan kelas CubeIterator yang memiliki metode __iter__() dan __next__(), di mana nilai dihitung dan dikembalikan satu per satu hingga mencapai batas n. Sementara itu, generator menggunakan fungsi cube_generator(n) dengan kata kunci yield, yang secara otomatis menangani status iterasi tanpa perlu menyimpan posisi saat ini secara eksplisit. Kedua pendekatan menghasilkan output yang sama, tetapi generator lebih ringkas dan lebih mudah digunakan karena tidak memerlukan implementasi metode __iter__() dan __next__().

Fungsi Iterator

```
class CubeIterator:
    def __init__(self, n):
        self.n = n
        self.current = 1

    def __iter__(self):
        return self

    def __next__(self):
        if self.current > self.n:
            raise StopIteration
        result = self.current ** 3
        self.current += 1
        return result
```

Hasil

```
n = int(input("Masukkan nilai n: "))
for num in CubeIterator(n):
    print(num)
```

```
Masukkan nilai n: 5
1
8
27
64
125
```

Fungsi Generator

```
def cube_generator(n):
    for i in range(1, n + 1):
        yield i ** 3
```

Hasil

```
n = int(input("Masukkan nilai n: "))
for num in cube_generator(n):
    print(num)
```

```
Masukkan nilai n: 5
1
8
27
64
125
```

3.3.2 Studi kasus 2 : Mengimplementasikan iterator dan generator yang menghasilkan angka acak dalam rentang tertentu.

Penjelasan:

Iterator kita buat menggunakan kelas RandomIterator, yang menyimpan batas bawah, batas atas, dan jumlah angka yang akan dihasilkan. Metode __next__() memastikan iterasi berhenti setelah mencapai jumlah yang ditentukan menggunakan StopIteration. Sementara itu, generator menggunakan fungsi random_generator(), yang lebih ringkas karena hanya memanfaatkan perulangan for dengan yield untuk mengembalikan angka acak tanpa menyimpan status iterasi secara manual.

Fungsi Iterator

```
import random

class RandomIterator:
    def __init__(self, start, end, count):
        self.start = start
        self.end = end
        self.count = count
        self.current = 0

    def __iter__(self):
        return self

    def __next__(self):
        if self.current >= self.count:
            raise StopIteration
        self.current += 1
        return random.randint(self.start, self.end)
```

Hasil

```
start = int(input("Masukkan batas bawah: "))
end = int(input("Masukkan batas atas: "))
count = int(input("Masukkan jumlah angka: "))

for num in RandomIterator(start, end, count):
    print(num)
```

```
Masukkan batas bawah: 5
Masukkan batas atas: 10
Masukkan jumlah angka: 6
9
8
5
6
7
8
```

Fungsi Generator

```
import random

def random_generator(start, end, count):
    for _ in range(count):
        yield random.randint(start, end)
```

Hasil

```
start = int(input("Masukkan batas bawah: "))
end = int(input("Masukkan batas atas: "))
count = int(input("Masukkan jumlah angka: "))

for num in random_generator(start, end, count):
    print(num)
```

```
Masukkan batas bawah: 5
Masukkan batas atas: 10
Masukkan jumlah angka: 6
10
7
10
10
5
9
```

3.3.3 Studi kasus 3 : Seorang peneliti ingin menganalisis suhu harian suatu kota selama satu tahun (365 hari). Karena keterbatasan memori, ia tidak ingin menyimpan seluruh data suhu dalam bentuk list, melainkan hanya ingin mengambil satu per satu nilai suhu saat diperlukan.

1) Buat sebuah iterator bernama TemperatureIterator yang menghasilkan suhu acak antara 20°C hingga 35°C selama 365 hari.
2) Buat sebuah generator bernama temperature_generator() yang menghasilkan nilai suhu dengan cara yang sama.

Penjelasan:

Berikut adalah hasil yang menunjukkan perbandingan antara iterator dan generator dalam mensimulasikan suhu harian selama satu tahun (365 hari). Iterator diimplementasikan melalui kelas TemperatureIterator, yang menyimpan jumlah hari dan status hari saat ini. Metode __next__() akan menghasilkan nilai suhu acak dalam rentang 20°C - 35°C setiap kali dipanggil, hingga mencapai jumlah hari yang ditentukan, lalu menghentikan iterasi dengan StopIteration. Sementara itu, generator menggunakan fungsi temperature_generator(), yang lebih sederhana.

Fungsi Iterator

```
import random

class TemperatureIterator:
    def __init__(self, days):
        self.days = days
        self.current_day = 0

    def __iter__(self):
        return self

    def __next__(self):
        if self.current_day >= self.days:
            raise StopIteration
        self.current_day += 1
        return random.uniform(20, 35) #20°C - 35°C
```

Hasil

```
# Menggunakan iterator
temp_iter = TemperatureIterator(365)
for temp in temp_iter:
    print(temp)  # Menampilkan suhu harian
```

```
20.98651556608526
22.765362848255318
21.008145534732876
32.84243827297841
27.80784346038506
26.055074471151887
31.764280788600054
22.501709391630733
33.857933957410644
```

Fungsi Generator

```
def temperature_generator(days):
    for _ in range(days):
        yield random.uniform(20, 35)
```

Hasil

```
# Menggunakan generator
temp_gen = temperature_generator(365)
for temp in temp_gen:
    print(temp)  # Menampilkan suhu harian
```

```
22.739251332855932
27.693349028540233
33.08423125076668
23.731605388039196
26.16800023422904
29.107947132632365
21.892012827611232
```

3.3.4 Studi kasus 3 : Data temperatur harian dalam file CSV berisi catatan suhu selama satu bulan. Tujuan kita adalah:

1) Menggunakan iterator untuk membaca dan memproses data suhu satu per satu.
2) Menggunakan generator untuk mengambil suhu di atas ambang batas tertentu.

Penjelasan:

- Iterator diimplementasikan melalui kelas TemperatureIterator, yang membuka file CSV, membaca isinya dengan csv.reader, dan melewati baris header. Metode __next__() akan mengembalikan tanggal dan suhu dari setiap baris hingga mencapai akhir file, lalu menutup file dan menghentikan iterasi dengan StopIteration.

- Generator diimplementasikan dalam fungsi temperature_generator(), yang juga membaca file CSV tetapi hanya mengembalikan data suhu yang lebih tinggi dari ambang batas (threshold). Ini memungkinkan filtering langsung dalam proses iterasi.

Fungsi Iterator

```python
import csv

class TemperatureIterator:
    def __init__(self, filename):
        self.filename = filename
        self.file = open(filename, mode='r')
        self.reader = csv.reader(self.file)
        next(self.reader)  # Lewati header

    def __iter__(self):
        return self

    def __next__(self):
        row = next(self.reader, None)
        if row is None:
            self.file.close()
            raise StopIteration
        return row[0], int(row[1])  # Kembalikan
```

Hasil

```python
# Menggunakan iterator
iterator = TemperatureIterator("temperature_data.csv")
for date, temp in iterator:
    print(f"{date}: {temp}°C")
```

```
2024-02-01: 27°C
2024-02-02: 30°C
2024-02-03: 25°C
2024-02-04: 28°C
2024-02-05: 32°C
2024-02-06: 31°C
2024-02-07: 26°C
2024-02-08: 29°C
```

Fungsi Generator

```python
def temperature_generator(filename, threshold):
    with open(filename, mode='r') as file:
        reader = csv.reader(file)
        next(reader)  # Lewati header
        for row in reader:
            date, temp = row[0], int(row[1])
            if temp > threshold:
                # Hanya kembalikan suhu di atas threshold
                yield date, temp
```

Hasil

```python
# Menggunakan generator dengan batas suhu 28°C
threshold = 28
for date, temp in temperature_generator("temperature_data.csv",
                                        threshold):
    print(f"Suhu tinggi pada {date}: {temp}°C")
```

```
Suhu tinggi pada 2024-02-02: 30°C
Suhu tinggi pada 2024-02-05: 32°C
Suhu tinggi pada 2024-02-06: 31°C
Suhu tinggi pada 2024-02-08: 29°C
```

Pada hasilnya, iterator menampilkan semua data suhu harian, sementara generator hanya mencetak suhu yang lebih tinggi dari 28°C, membuatnya lebih efisien untuk analisis data dengan kriteria tertentu.

3.4 Tugas Individu