**LAPORAN PRAKTIKUM**

**STRUKTUR DATA**

**MODUL KE-08**

**GRAPH DALAM PYTHON**



**Disusun Oleh:**

**Nama :** Restu Wibisono

**NPM :** 2340506061

**Kelas :** 03 (Tiga)

**Program Studi S1 Teknologi Informasi**

**Fakultas Teknik, Universitas Tidar**

**Genap 2023/2024**

1. **Tujuan Praktikum**

Adapun tujuan praktikum ini sebagai berikut :

* 1. Mahasiswa mampu menerapkan konsep graph pada bahasa pemrograman python

1. **Dasar Teori (minimal 3 halaman)**

Struktur Data Graf adalah kumpulan simpul yang terhubung oleh tepi. Digunakan untuk merepresentasikan hubungan antara entitas yang berbeda. Algoritma graf adalah metode yang digunakan untuk memanipulasi dan menganalisis graf, memecahkan berbagai masalah seperti menemukan lintasan terpendek atau mendeteksi siklus.

Graf adalah struktur data non-linear yang terdiri dari simpul dan tepi. Simpul kadang-kadang juga disebut sebagai simpul dan tepi adalah garis atau lengkungan yang menghubungkan dua simpul dalam graf. Lebih formal sebuah Graf terdiri dari kumpulan simpul (V) dan kumpulan tepi (E). Graf tersebut dilambangkan dengan G(V, E).

Komponen Graf:

* Simpul: Simpul adalah unit dasar dari graf. Kadang-kadang, simpul juga dikenal sebagai simpul atau simpul. Setiap simpul/simpul dapat diberi label atau tidak diberi label.
* Tepi: Tepi digambar atau digunakan untuk menghubungkan dua simpul graf. Ini bisa berupa pasangan terurut dari simpul dalam graf terarah. Tepi dapat menghubungkan dua simpul dalam cara apa pun. Tidak ada aturan. Kadang-kadang, tepi juga dikenal sebagai lengkungan. Setiap tepi dapat diberi label/tanpa label.

Berikut adalah operasi dasar pada graf:

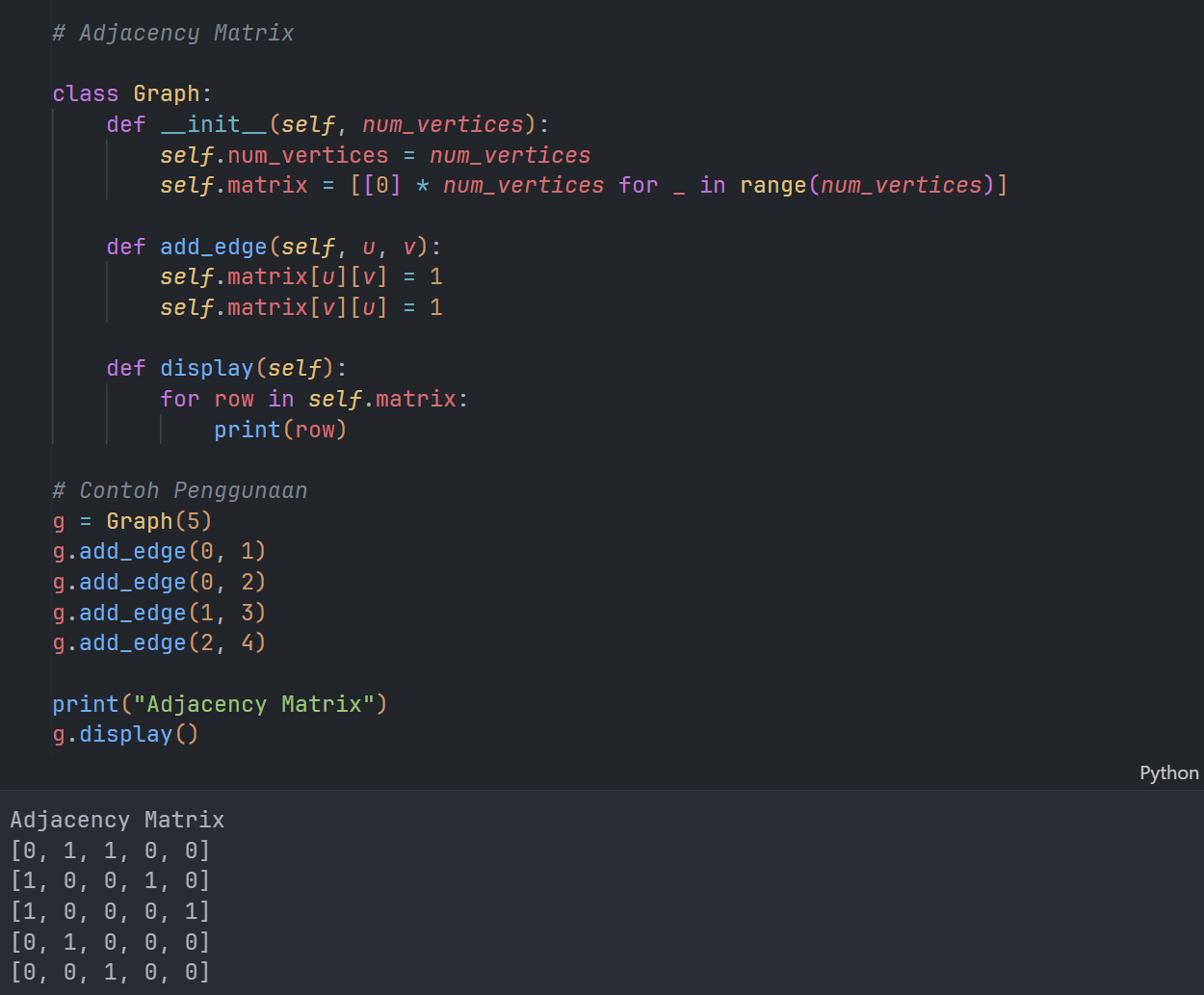
* Penyisipan Simpul/Tepi dalam graf - Memasukkan simpul ke dalam graf.
* Penghapusan Simpul/Tepi dalam graf - Menghapus simpul dari graf.
* Pencarian pada Graf - Mencari entitas dalam graf.
* Traversal Graf - Melintasi semua simpul dalam graf.

|  |  |
| --- | --- |
| Kelebihan Graph | Kekurangan Graph |
| * Graf dapat digunakan untuk memodelkan dan menganalisis sistem dan hubungan yang kompleks. * Graf bermanfaat untuk memvisualisasikan dan memahami data. * Algoritma graf secara luas digunakan dalam ilmu komputer dan bidang lainnya, seperti analisis jaringan sosial, logistik, dan transportasi. * Graf dapat digunakan untuk merepresentasikan berbagai jenis data, termasuk jaringan sosial, jaringan jalan, dan internet | * Graf yang besar dapat sulit untuk divisualisasikan dan dianalisis. * Algoritma graf dapat memakan banyak sumber daya komputasi, terutama untuk graf yang besar. * Interpretasi hasil graf dapat subjektif dan mungkin memerlukan pengetahuan spesifik domain. * Graf dapat rentan terhadap noise dan outliers, yang dapat memengaruhi akurasi hasil analisis. |

Perbedaan Kunci Antara Graf dan Pohon:

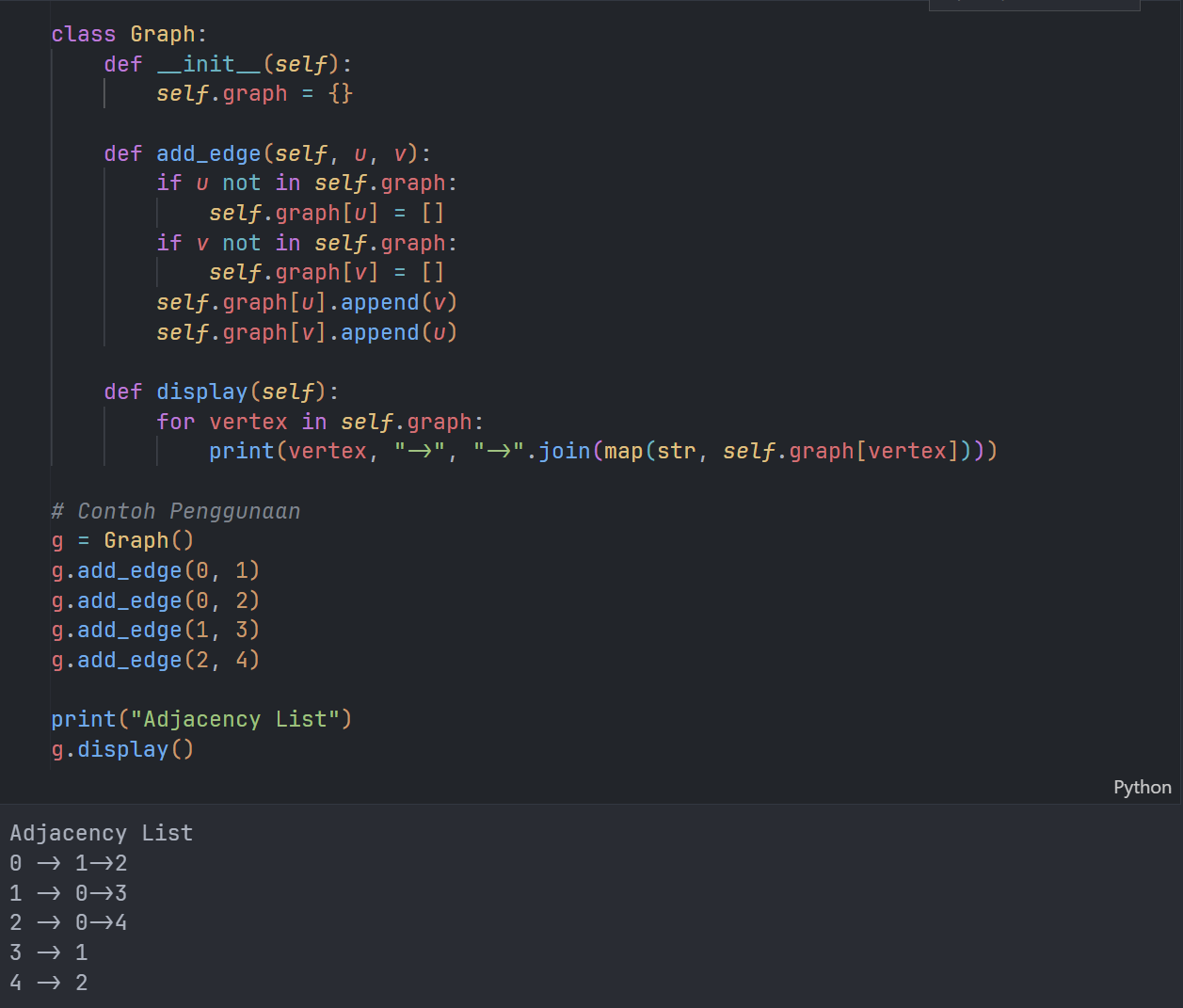
* Siklus: Graf dapat mengandung siklus, sedangkan pohon tidak bisa.
* Konektivitas: Graf bisa tidak terhubung (misalnya, memiliki beberapa komponen), sedangkan pohon selalu terhubung.
* Hirarki: Pohon memiliki struktur hirarkis, dengan satu titik ditetapkan sebagai akar. Graf tidak memiliki struktur hirarkis ini.
* Aplikasi: Graf digunakan dalam berbagai aplikasi, seperti jaringan sosial, jaringan transportasi, dan ilmu komputer. Pohon sering digunakan dalam struktur data hirarkis, seperti sistem file dan dokumen XML.

1. **Hasil dan Pembahasan**
2. Adjacency Matrix

****

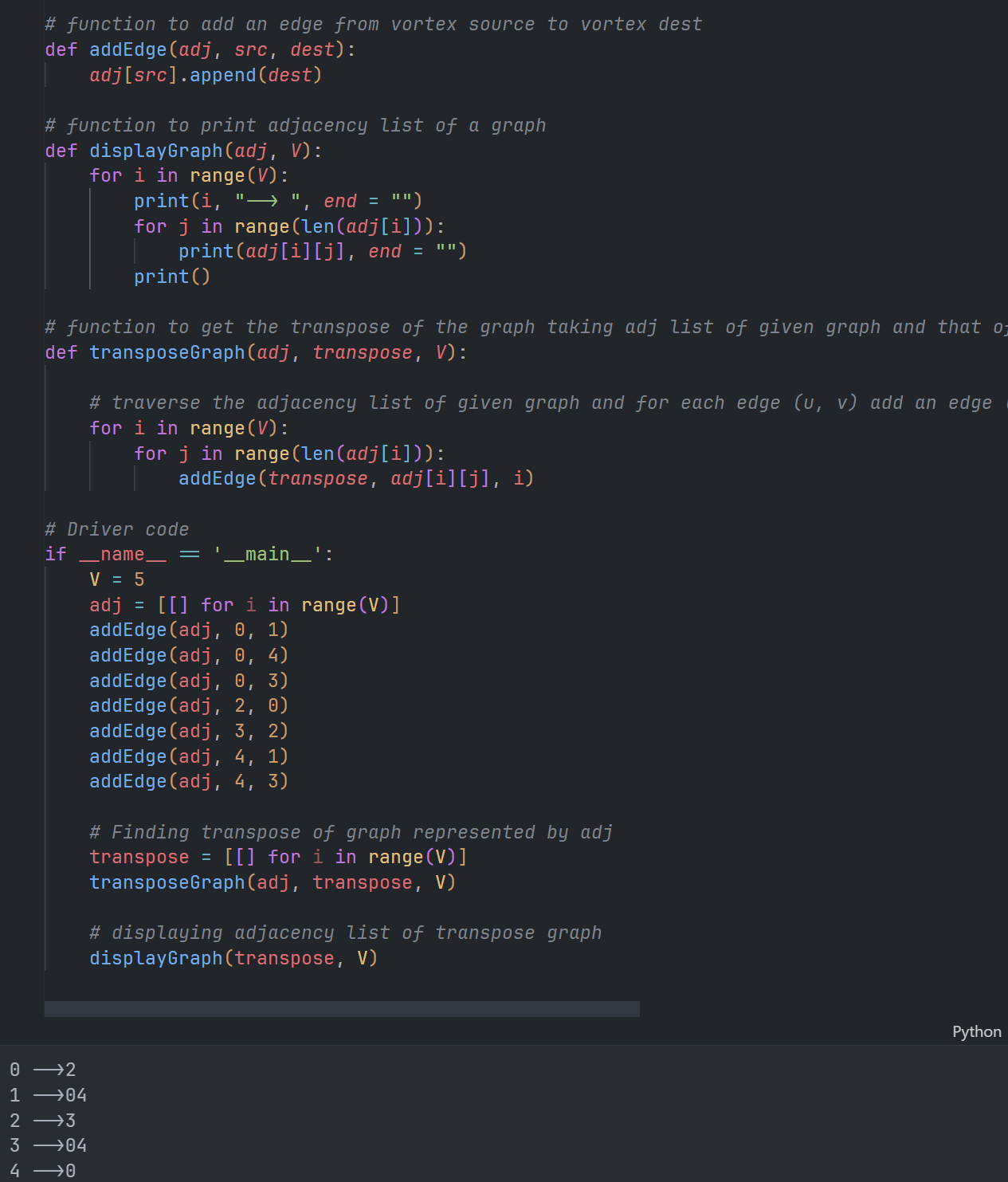
(Gambar 3.1.1)

1. Awal dari definisi kelas yang di sebut `Graph`, yang digunakan untuk merepresentasikan graf.
2. Metode khusus yang disebut saat objek `Graph` dibuat, `num\_vertices` untuk parameter yang menentukan jumlah simpul dalam graf.
3. `self.num\_vertices = num\_vertices` berfungsi untuk menginisialisasi atribut `num\_vertices` dari objek `Graph`.
4. Membuat matriks dengan `self.matrix` dengan ukuran `num\_vertices x num\_vertices`. Yang akan di gunakan untuk menyimpan informasi hubungan antar simpul.
5. Menambahkan tepian (edge) antara dua simpul `u` dan `v` dalam graf dengan `def add\_edge(self, u, v):`.
6. Untuk menetapkan nilai 1 dalam posisi `(u, v)` dan `(v, u)` dalam matriks `self.matrix[u][v] = 1` dan `self.matrix[v][u] = 1` untuk menunjukkan terdapat tepian antara simpul `u` dan `v`.
7. Dengan `def display(self):` untuk menampilkan matriks keberadjadian ke layar.
8. Program `for row in self.matrix:` Melakukan iterasi pada setiap baris dalam matriks.
9. `print(row)` akan mencetak setiap baris matriks.
10. AdjacencyList

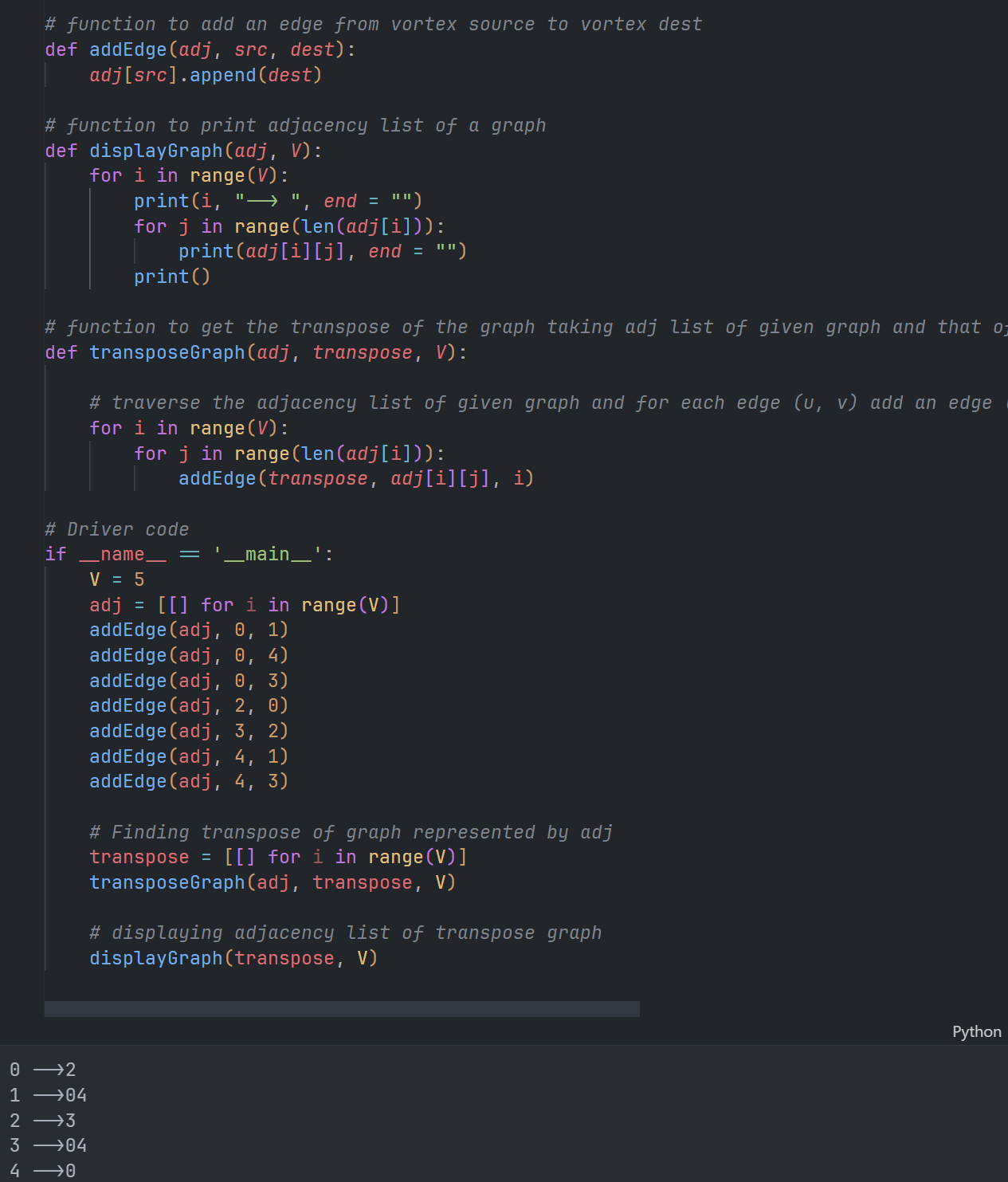
****

(Gambar 3.2.1)

1. Pertama membuat `class Graph:` dari definisi kelas yang disebut `Graph`, yang akan digunakan untuk merepresentasikan graf.
2. Melakukan metode khusus dengan `def \_\_init\_\_(self):` yang disebut saat objek `Graph` dibuat.
3. Membuat sebuah dictionary `self.graph = {}` yang akan berfungsi untuk menyimpan daftar keberadjadian setiap simpul dalam graf.
4. Menambahkan tepian (edge) antara dua simpul `u` dan `v` dalam graf dengan `def add\_edge(self, u, v):`.
5. Program akan memeriksa apakah simpul `u` sudah ada dalam kamus `if u not in self.graph:` jika tidak, maka di buat entri baru untuk simpul `u`.
6. Lalu memeriksa apakah simpul `v` sudah ada dalam kamus `if v not in self.graph:` jika tidak, maka di buat entri baru untuk simpul `v`.
7. Selanjutnya menambahkan simpul `v` ke daftar keberadjadian simpul `u`, menunjukkan bahwa ada tepian dari `u` ke `v` dengan `self.graph[u].append(v)`.
8. Menambahkan simpul `u` ke dalam daftar keberadjadian simpul `v`, untuk menunjukkan bahwa ada tepian dari `v` ke `u`dengan `self.graph[v].append(u)`.
9. Menampilkan daftar keberadjadian ke layar dengan metode `def display(self):`.
10. Melakukan iterasi pada setiap simpul `for vertex in self.graph:` dalam kamus.
11. Terakhir mencetak simpul `print(vertex, "->", "->".join(map(str, self.graph[vertex])))` yang diikuti oleh panah (->) dan daftar keberadjadiannya.
12. Transpose Graf

****

(Gambar 3.3.1)

****

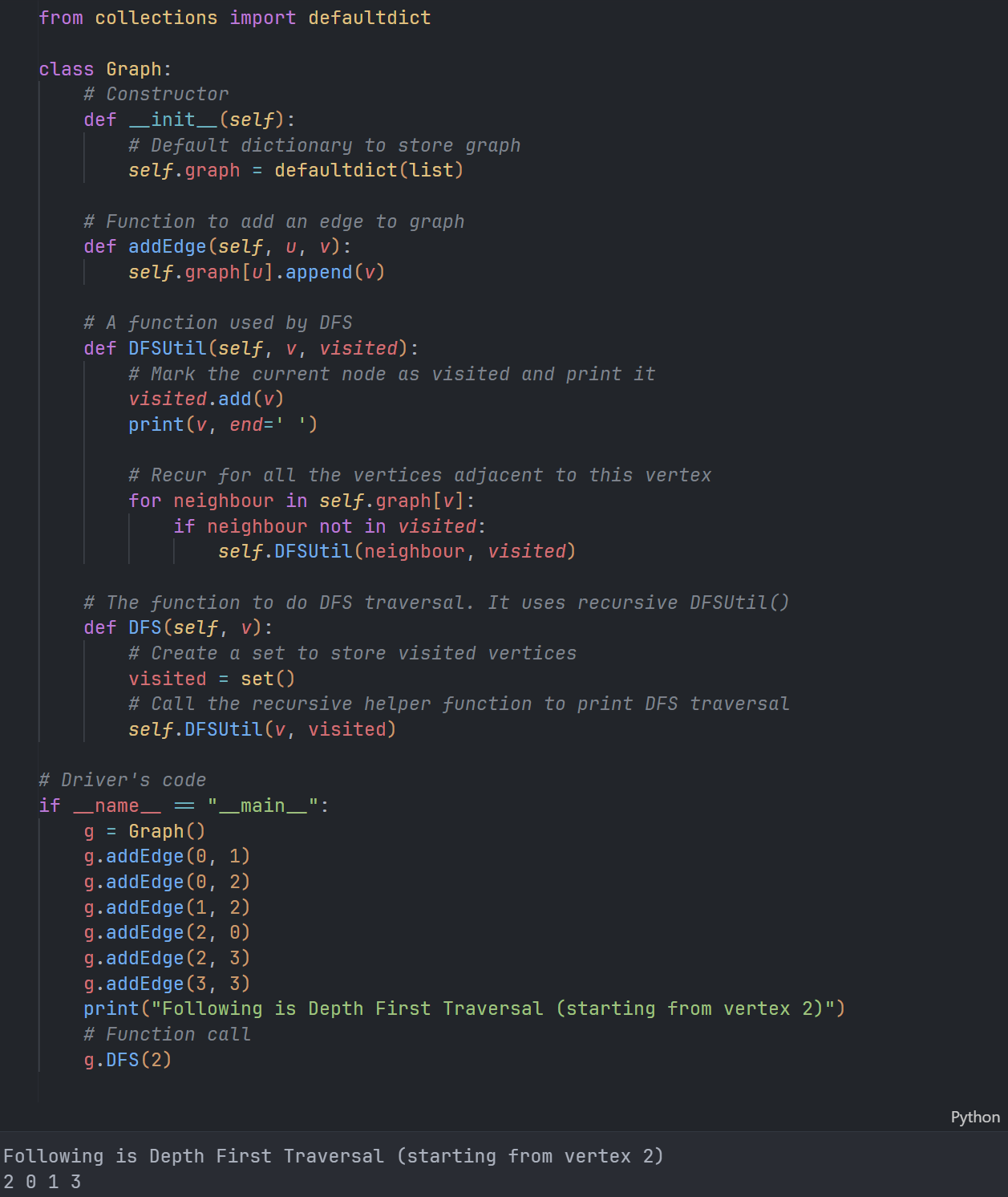
(Gambar 3.3.2)

1. Program dimulai dengan `def addEdge(adj, src, dest):` yang berfungsi untuk mendefinisi sebuah fungsi bernama `addEdge` yang akan menerima tiga parameter: `adj`, `src`, dan `dest`. Dimana fungsi ini untuk menambahkan sebuah tepian dari simpul `src` ke `dest` dalam daftar keberadjadian `adj`.
2. Menambahkan simpul `adj[src].append(dest)` untuk menambahkan `dest` ke daftar keberadjadian `src`, lalu menunjukkan adanya tepian simpul `src` ke `dest`.
3. Mendefinisikan sebuah fungsi dengan `def displayGraph(adj, V):` bernama `displayGraph` untuk menerima dua parameter: `adj`) dan `V` ini berfungsi untuk menampilkan daftar keberadjadian graf.
4. Melakukan iterasi pada setiap simpul dalam graf dengan `for i in range(V):`.
5. `print(i, "--> ", end = "")` Mencetak nomor simpul yang diikuti oleh tanda panah (-->). Penggunaan `end = ""` akan memastikan cetakan berikutnya tidak pergi ke baris baru.
6. Melakukan iterasi `for j in range(len(adj[i])):` untuk setiap simpul yang berdekatan dengan simpul dalam daftar keberadjadian.
7. Mencetak simpul yang berdekatan dengan simpul saat ini dengan metode `print(adj[i][j], end = "")`.
8. `print()` berfungsi utnuk mencetak baris baru setelah selesai mencetak semua simpul.
9. Mendefinisikan dengan `def transposeGraph(adj, transpose, V):` untuk sebuah fungsi bernama `transposeGraph` yang akan berfungsi menerima tiga parameter.
10. `for i in range(V):` Melakukan iterasi pada setiap simpul di dalam graf.
11. `for j in range(len(adj[i])):` Melakukan iterasi untuk setiap simpul yang berdekatan dengan simpul saat ini dalam daftar keberadjadian graf asli.
12. Memanggil fungsi `addEdge` berfungsi menambahkan tepian simpul yang dekat dengan simpul saat ini menuju simpul saat ini.
13. `if \_\_name\_\_ == '\_\_main\_\_':` akalah blok kode yang akan di jalankan saat skrip di eksekusi dengan langsung.
14. Menetapkan jumlah simpul dalam graf dengan `V = 5`.
15. Membuat daftar kosong `adj = [[] for i in range(V)]` untuk menyimpan keberadjadian dari setiap simpul dalam graf.
16. Menambahkan beberapa tepian ke graf menggunakan fungsi `addEdge`.
17. Membuat daftar kosong `transpose = [[] for i in range(V)]` untuk menyimpan keberadjadian transpose graf.
18. Dengan `transposeGraph(adj, transpose, V)` akan memanggil fungsi `transposeGraph` yang ebrfungsi untuk menghitung transpose dari graf.
19. Menampilkan daftar keberadjadian dari graf transpose `displayGraph(transpose, V)`.
20. Breadth Frist Search (BFS)

****

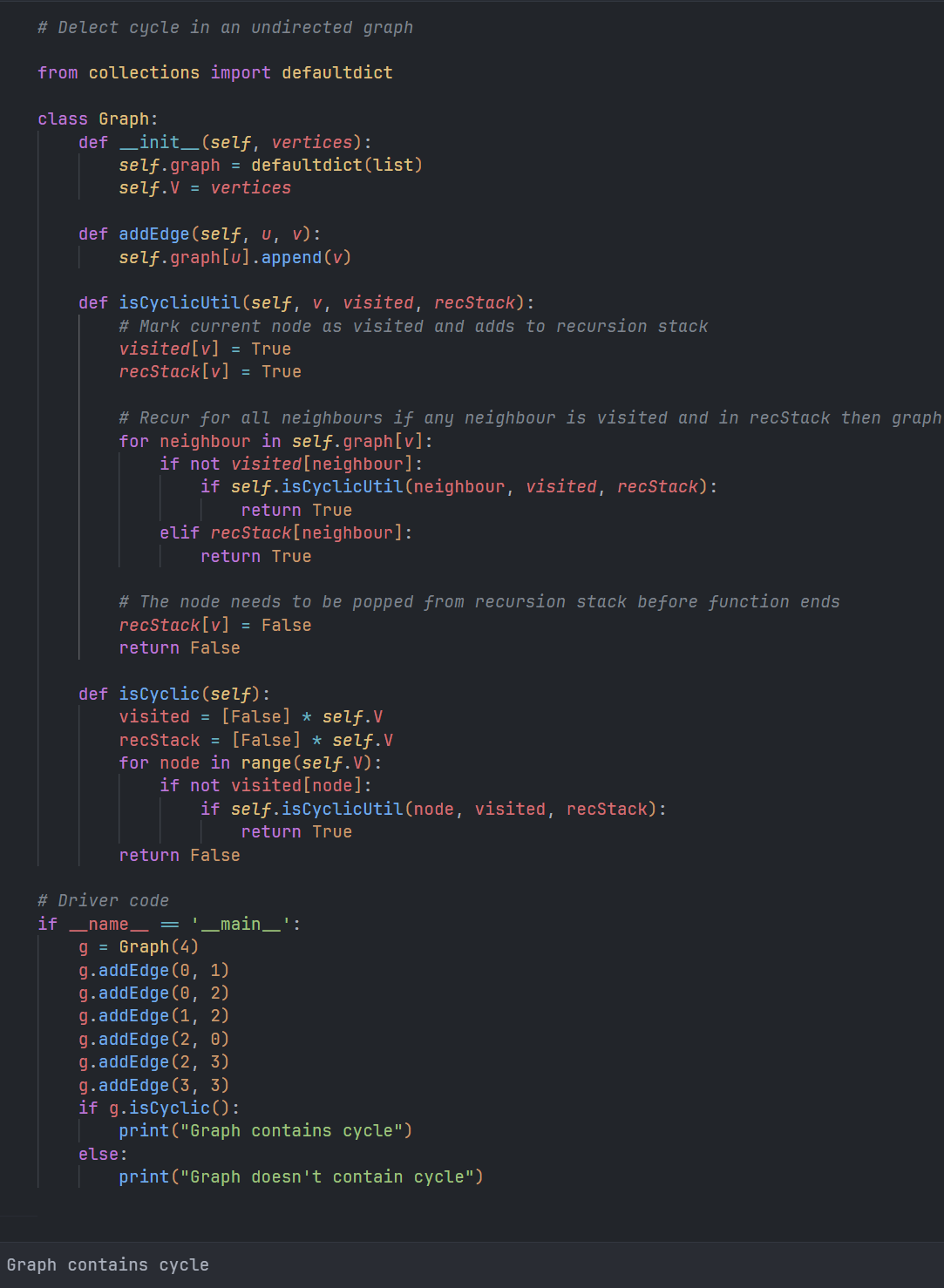
(Gambar 3.4.1)

1. Pertama `from collections import deque` yang berfungsi untuk mengimpor kelas `deque` dari modul `collection` di gunakan sebagai antrian dalam algoritma BFS.
2. Selanjutnya `def bfs(adjList, startNode, visited):` adalah definisi untuk fungsi `bfs` yang menerima tiga parameter.
3. `q = deque()`: membuat objek untuk antrian `q` menggunakan `deque()`.
4. Menandai simpul awal `startNode` sebagai telah dikunjungi.
5. Memasukkan simpul awal ke dalam antrian dengan metode `q.append(startNode)`.
6. Melakukan loop `while q:` selama antrian tidak kosong.
7. Menghapus dan mengembalikan elemen dengan `currentNode = q.popleft()`dari antrian, yang merupakan fungsi untuk simpul yang sedang di proses.
8. Mencetak simpul `print(currentNode, end=" ")` yang sedang diproses.
9. Melakukan iterasi pada semua tetangga `for neighbor in adjList[currentNode]:` dari simpul yang sedang di proses.
10. Memeriksa `if not visited[neighbor]:` apakah tetangga belum dikunjungi.
11. Menandai tetangga `visited[neighbor] = True` tersebut sebagai telah dikunjungi.
12. Memasukkan tetangga `q.append(neighbor)` yang belum dikunjungi ke dalam antrian.
13. Mendefinisikan dari fungsi `addEdge` yang di gunakan untuk menambahkan tepian baru ke daftar keberadjadian `adjList`.
14. Menetapkan jumlah simpul dalam graf dengan metode `vertices = 5`.
15. Membuat daftar kosong `adjList = [[] for \_ in range(vertices)]` yang berfungsi untuk menyimpan daftar keberadjadian simpul dalam graf.
16. Selanjutnya menambahkan beberapa tepian ke graf menggunakan fungsi `addEdge`.
17. Membuat daftar `visited` yang menandai semua simpul belum dikunjungi.
18. Mencetak pesan untuk menandai bahwa pencarian BFS dimulai dari simpul 0 dengan metode `print("Breadth First Traversal starting from vertex 0:", end=" ")`.
19. Memanggil fungsi ``bfs(adjList, 0, visited)` untuk melakukan pencarian BFS dari simpul 0.
20. Blok kode `if \_\_name\_\_ == "\_\_main\_\_":` yang akan di jalankan saat skrip di eksekusi secara langsung.
21. Memanggil fungsi `main` untuk memulai eksekusi program dengan `main()`.
22. Depth Frist Search (DFS)

****

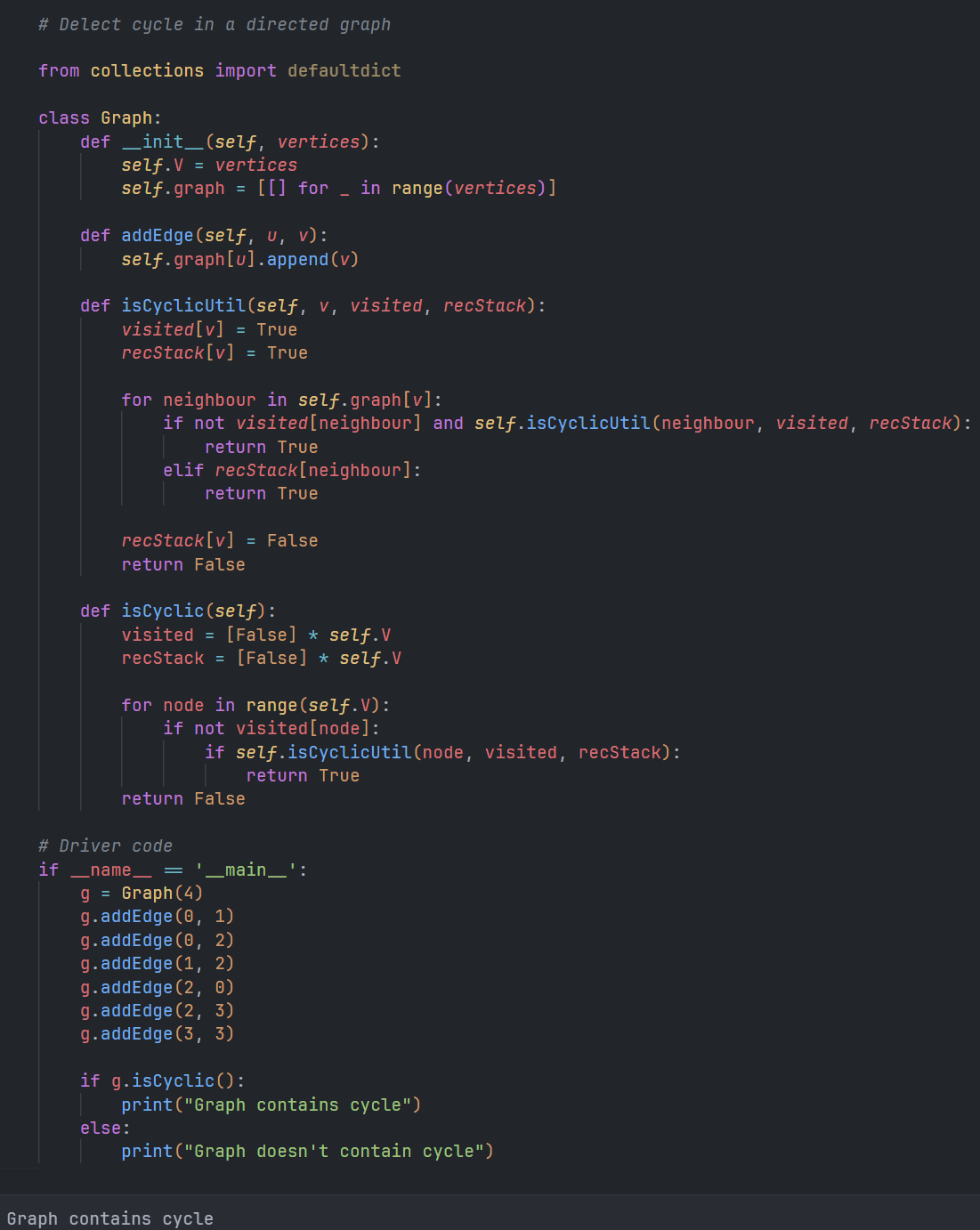
(Gambar 3.5.1)

1. Mengimpor kelas `defaultdict` dari modul `collections`. `defaultdict` berfungsi di sini untuk membuat struktur data graf yang lebih mudah.
2. Membuat definisi `class Graph:`, yang akan di gunakan untuk merepresentasikan graf.
3. `def \_\_init\_\_(self):` berfungsi menjadi metode konstruktor kelas `Graph`.
4. Membuat dictionary `self.graph = defaultdict(list)` yang menggunakan `defaultdict` berfungsi untuk menyimpan daftar keberadjadian dari setiap simpul.
5. Metode `def addEdge(self, u, v):` berfungsi untuk menambahkan tepian (edge) antara dua simpul `u` dan `v` dalam graf.
6. Menambahkan simpul `v` ke daftar keberadjadian simpul `u` dengan `self.graph[u].append(v)`.
7. Metode utilitas `def DFSUtil(self, v, visited):` yang digunakan oleh DFS.
8. Untuk menandai simpul `visited.add(v)` saat ini sebagai telah dikunjungi.
9. `print(v, end=' ')` Mencetak simpul saat ini.
10. Melakukan iterasi `for neighbour in self.graph[v]:` untuk semua tetangga dari simpul saat ini dalam daftar keberadjadian.
11. Memeriksa `if neighbour not in visited:` apakah tetangga belum dikunjungi.
12. Memanggil `self.DFSUtil(neighbour, visited)` rekursif untuk menjelajahi tetangga yang belum di kunjungi.
13. Metode `def DFS(self, v):` untuk melakukan DFS traversal dari simpul `v`.
14. Lalu membuat himpunan `visited = set()` untuk menyimpan simpul yang telah di kunjungi.
15. Memanggil `self.DFSUtil(v, visited)` metode utilitas DFS untuk memulai traversal dari simpul `v`.
16. Dengan blok kode `if \_\_name\_\_ == "\_\_main\_\_":` yang akan di jalankan saat skrip di eksekusi secara langsung.
17. Membuat objek `g` dari kelas `Graph`.
18. Menambahkan beberapa edge ke graf menggunakan metode `addEdge`.
19. Mencetak pesan untuk menandai pencarian DFS yang di mulai dari simpul dengan metode `print("Following is Depth First Traversal (starting from vertex 2)")`.
20. Memanggil metode `DFS` untuk memulai DFS traversal dari simpul
21. Siklus Detect Cycle in a Directed Graph

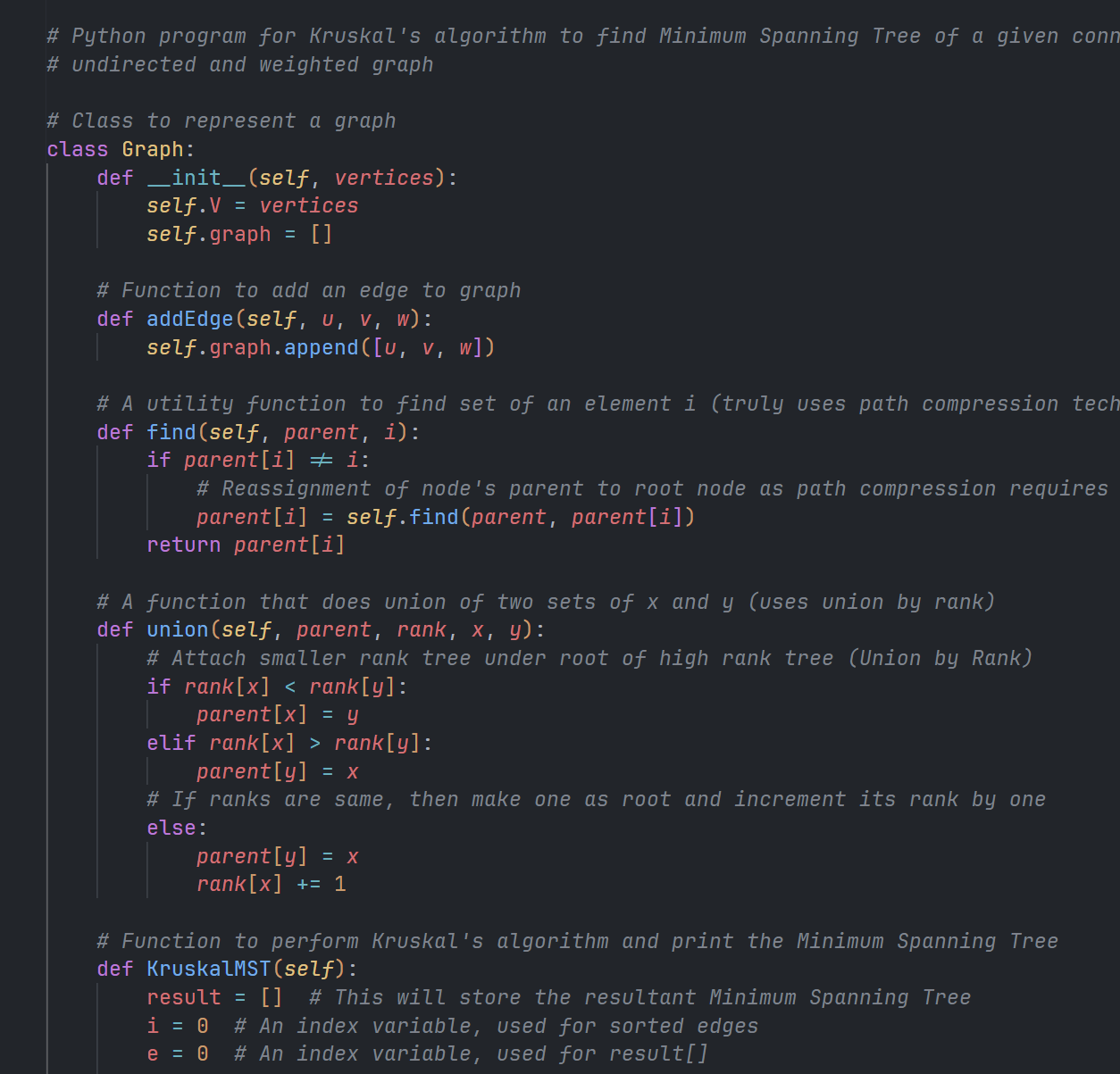
****

(Gambar 3.6.1)

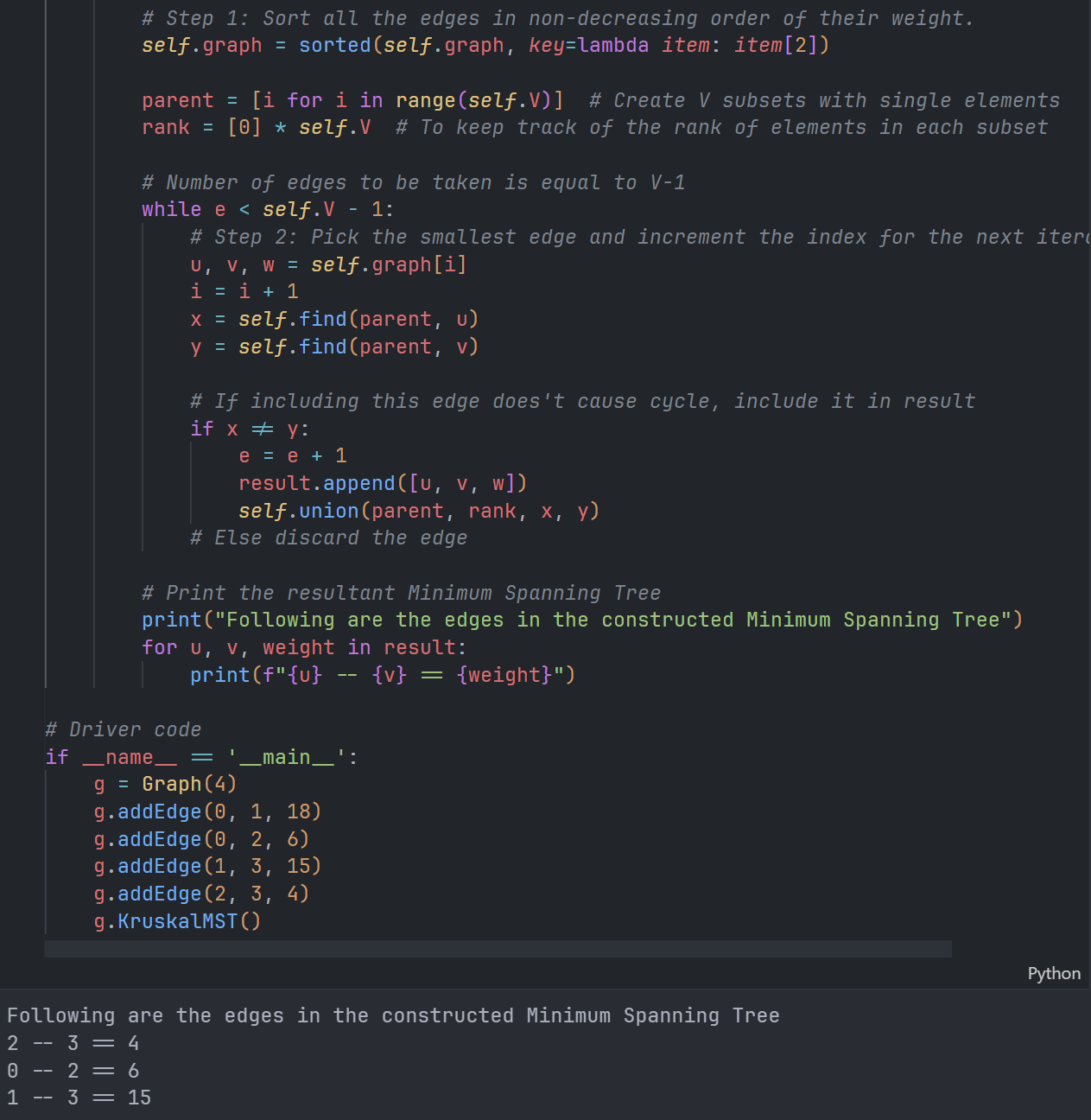
1. Mengimpor kelas `from collections import defaultdict` dari modul `collections`, `defaultdict` digunakan membuat struktur graf.
2. Mendefinisikan kelas dengan `class Graph:`, yang akan di gunakan untuk merepresentasikan graf.
3. Selanjutnay `def \_\_init\_\_(self, vertices):` berfungsi untuk konstruktor kelas `Graph` yang akan menerima parameter `vertices`.
4. Membuat dictionary `self.graph = defaultdict(list)` yang menggunakan `defaultdict` berfungi untuk menyimpan daftar keberadjadian setiap simpul dalam graf.
5. Menyimpan jumlah simpul dalam atribut `V` dengan `self.V = vertices`.
6. `def addEdge(self, u, v):` untuk menambahkan tepian antara dua simpul `u` dan `v` dalam graf.
7. Menambahkan simpul `v` ke daftar keberadjadian simpul `u` dengan `self.graph[u].append(v)`.
8. Metode utilitas `def isCyclicUtil(self, v, visited, recStack):` yang di gunakan untuk mendeteksi siklus dengan cara menggunakan rekursif.
9. Menandai simpul saat ini dengan `visited[v] = True` dan `recStack[v] = True` sebagai telah di kunjungi dan menambahkannya ke dalam tumpukan rekursi.
10. Program `def isCyclic(self):` adalah metode untuk memeriksa apakah graf berisi siklus atau tidak.
11. Lalu jika `isCyclicUtil` mengembalikan `True`, maka dalam graf berisi siklus dan metode mengembalikan `True`.
12. Blok kode `if \_\_name\_\_ == '\_\_main\_\_':` yang akan dijalankan jika skrip di eksekusi secara langsung.
13. Selanjutnya menambahkan beberapa edge ke graf menggunakan metode `addEdge`.
14. Memanggil metode `isCyclic` yang berfungsi untuk memeriksa apakah dalam graf berisi siklus atau tidak, dan mencetak hasilnya.
15. Siklus Detect cycle in an undirected graph

****(Gambar 3.7.1)

1. Pertama mengimpor kelas `from collections import defaultdict` dari modul `collections`, pada `defaultdict` di gunakan untuk membuat struktur data graf yang lebih mudah.
2. Selanjutnya mendefinisi dari kelas `Graph`, yang akan digunakan untuk merepresentasikan graf dengan merode `class Graph:`.
3. Pada metode konstruktor kelas `Graph` yang menerima parameter `def \_\_init\_\_(self, vertices):`, yang akan menunjukkan jumlah simpul dalam graf.
4. `def addEdge(self, u, v):` adalah metode yang berfungsi untuk menambahkan tepian antara simpul `u` dan `v` dalam graf.
5. Menambahkan simpul `v` ke daftar keberadjadian simpul `u` dengan metode `self.graph[u].append(v)`.
6. Metode utilitas `def isCyclicUtil(self, v, visited, recStack):` yang berfungsi untuk mendeteksi siklus menggunakan rekursif.
7. Menandai simpul `visited[v] = True` dan `recStack[v] = True` saat ini telah di kunjungi dan menambahkannya dalam tumpukan rekursi.
8. Metode `def isCyclic(self):` untuk memeriksa pada graf berisi siklus atau tidak.
9. Blok kode `if \_\_name\_\_ == '\_\_main\_\_':` akan berfungsi disaat skrip dijalankan secara langsung.
10. Menambahkan beberapa edge ke graf dengan metode `addEdge`.
11. Memanggil metode `isCyclic` yang berfungsi untuk memeriksa apakah graf berisi siklus.
12. Kruskal’s algorithm to find Minimum Spanning Tree

****

(Gambar 3.8.1)

****

(Gambar 3.8.2)

1. Pertama membuat `class Graph:` definisi dari kelas `Graph`, yang akan berfunsgi untuk merepresentasikan graf.
2. Metode konstruktor `def \_\_init\_\_(self, vertices):` kelas `Graph` akan menerima parameter `vertices`, berfungsi menunjukkan jumlah simpul dalam graf.
3. Metode `def addEdge(self, u, v, w):` untuk menambahkan edge antara simpul `u` dan `v` dengan `w` ke dalam graf.
4. Selanjutnya metode utilitas `def find(self, parent, i):` yang berfungsi menemukan himpunan (set) dari elemen `i`.
5. Fungsi `def union(self, parent, rank, x, y):` adalah utilitas untuk melakukan penggabungan dua himpunan dengan teknik penggabungan berdasarkan peringkat.
6. Selanjutnya pada `def KruskalMST(self):` akan melakukan algoritma Kruskal dan mencetak Minimum Spanning Tree.
7. Blok kode `if \_\_name\_\_ == '\_\_main\_\_':` akan dijalankan saat skrip di eksekusi secara langsung.
8. Membuat objek `g` denagn kelas `Graph` dari 4 simpul.
9. Menambahkan beberapa edge ke dalam graf dengan menggunakan metode `addEdge`.
10. Memanggil metode `KruskalMST` yang akan mencetak MST dari graf.
11. **Latihan**

****

(Gambar 4.1)

1. `class Graph:` berfungsi untuk definisi kelas `Graph`, yang akan digunakan untuk merepresentasikan graf.
2. Dengan metode `def \_\_init\_\_(self, num\_vertices):` konstruktor kelas `Graph` akan menerima parameter `num\_vertices`, yang menunjukkan jumlah simpul dalam graf.
3. Selanjutnya metode `def add\_edge(self, u, v):` untuk menambahkan edge antara simpul `u` dan `v` ke dalam graf.
4. `def topological\_sort\_util(self, v, visited, stack):` adalah metode utilitas yang berfungsi untuk melakukan topological sort dengan menggunakan pendekatan rekursif.
5. Metode `def topological\_sort(self):` berfungi untuk melakukan topological sort pada graf.
6. Blok kode `if \_\_name\_\_ == "\_\_main\_\_":` yang akan berjalan saat skrip di eksekusi secara langsung.
7. Menetapkan jumlah simpul dalam graf `V = 5`.
8. Menetapkan tepian dengan `edges = [(0, 1), (0, 2), (1, 3), (2, 4), (3, 4)]` yang akan ditambahkan ke graf.
9. Membuat objek `graph` dari kelas `Graph` dengan 5 simpul.
10. Menambahkan edge ke graf menggunakan metode `add\_edge`.
11. Memanggil metode `topological\_sort` untuk mendapatkan urutan topological dari graf.
12. Mencetak urutan topological hasilnya.
13. **Kesimpulan**

Program di atas akan membantu Anda lebih memahami penggunaan struktur data dan algoritma dalam konteks grafik. Program ini menggunakan kelas Graph dan metodenya untuk mengimplementasikan algoritma pengurutan topologi menggunakan pendekatan depth-first search (DFS). DFS digunakan untuk menjelajahi grafik dan menentukan urutan topologi simpulnya, dan struktur data seperti daftar dan daftar dua dimensi digunakan untuk menyimpan informasi tentang grafik dan perilaku algoritma.

Implementasi algoritma pengurutan topologi ini memberikan pemahaman yang lebih mendalam tentang konsep pengurutan topologi, dimana setiap sisi grafik menghubungkan dari node sebelumnya ke node berikutnya. Dengan memahami implementasi ini, siswa akan memperoleh pemahaman lebih dalam tentang struktur data grafik, algoritma pengurutan topologi, dan penerapan DFS untuk menyelesaikan masalah grafik. Lokakarya yang mencakup latihan-latihan ini merupakan langkah efektif untuk memperdalam pemahaman Anda tentang konsep-konsep ini.