15.3 Representing Graphs in Python

Now that we've learned about some basic definitions and properties about graphs, let's see how we can represent graphs in Python. The approach we'll take in this section is to implement graphs as a *node-based data structure*, just like we did with linked lists all the way back in Chapter 11. Intuitively, each *vertex* of a graph will correspond to a "node object", and the edges of a graph will be represented implicitly by having each node store references to its neighbours.

Let's jump into it!

The `_Vertex` class

Here is the start of a `_Vertex` class, which represents a single vertex in a graph.[1]

[1] Note the similarity between this class and the `_Node` class we developed for linked lists. We're again choosing to make our "node" class private, as we'll expect users not to interact with it directly.

```
from __future__ import annotations
from typing import Any


class _Vertex:
    """A vertex in a graph.

    Instance Attributes:
        - item: The data stored in this vertex.
        - neighbours: The vertices that are adjacent to this vertex.
    """
    item: Any
    neighbours: set[_Vertex]

    def __init__(self, item: Any, neighbours: set[_Vertex]) -> None:
        """Initialize a new vertex with the given item and neighbours."""
        self.item = item
        self.neighbours = neighbours
```

This definition is fairly straightforward: the `item` instance attribute stores the "data" in each vertex, which may be an integer, a string, or a custom data type that we define. The `neighbours` attribute is how we encode the graph edges: it stores a set of other `_Vertex` objects. This attribute is structurally similar to the `Tree` `_subtrees` attribute, but now we're back to treating the class as representing a single element of the graph, rather than the

whole graph itself. Also note that this collection is *unordered*: we don't have a notion of "left-to-right" ordering like we did we trees.

For example, here is how we could represent a "triangle": three vertices that are all adjacent to each other.

```
>>> v1 = _Vertex('a', set())
>>> v2 = _Vertex('b', set())
>>> v3 = _Vertex('c', set())
>>> v1.neighbours = {v2, v3}
>>> v2.neighbours = {v1, v3}
>>> v3.neighbours = {v1, v2}
```

Enforcing edge restrictions

Recall that graph edges have two important restrictions: we cannot have a vertex with an edge to itself, and all edges are symmetric (that is, the edge $\{u, v\}$ is equivalent to the edge $\{v, u\}$). As you're probably expecting by now, we can enforce these properties by adding two representation invariants to our `_Vertex` class:

```
class _Vertex:
    """A vertex in a graph.

    Instance Attributes:
        - item: The data stored in this vertex.
        - neighbours: The vertices that are adjacent to this vertex.

    Representation Invariants:
        - self not in self.neighbours
        - all(self in u.neighbours for u in self.neighbours)
    """
    item: Any
    neighbours: set[_Vertex]
```

The `Graph` class

Next, we can define a `Graph` class that simply consists of a collection of `_Vertex` objects.[2]

[2] Remember, even though we defined a graph formally in the previous section as a collection of vertices *and* edges, in our Python representation the edges are stored implicitly as references in the `_Vertex` class.

```
class Graph:
    """A graph.
    """
    # Private Instance Attributes:
```

```
    #     - _vertices: A collection of the vertices contained in this graph.
    #                  Maps item to _Vertex instance.
    _vertices: dict[Any, _Vertex]

    def __init__(self) -> None:
        """Initialize an empty graph (no vertices or edges)."""
        self._vertices = {}
```

Similar to our other collection data types, our initializer is very restricted: we only create empty graphs. In order to build up a larger graph, we'll provide mutating methods to insert new vertices and edges. Here are two such methods:

```
class Graph:
    def add_vertex(self, item: Any) -> None:
        """Add a vertex with the given item to this graph.

        The new vertex is not adjacent to any other vertices.

        Preconditions:
            - item not in self._vertices
        """
        self._vertices[item] = _Vertex(item, set())

    def add_edge(self, item1: Any, item2: Any) -> None:
        """Add an edge between the two vertices with the given items in this
         graph.

        Raise a ValueError if item1 or item2 do not appear as vertices in
         this graph.

        Preconditions:
            - item1 != item2
        """
        if item1 in self._vertices and item2 in self._vertices:
            v1 = self._vertices[item1]
            v2 = self._vertices[item2]

            # Add the new edge
            v1.neighbours.add(v2)
            v2.neighbours.add(v1)
        else:
            # We didn't find an existing vertex for both items.
            raise ValueError
```

With this, we can create `Graph` objects and populate them with vertices and edges:

```
>>> g = Graph()
>>> g.add_vertex('a')
```

```
>>> g.add_vertex('b')
>>> g.add_vertex('c')
>>> g.add_edge('a', 'b')
>>> g.add_edge('a', 'c')
>>> g.add_edge('b', 'c')
```

Checking adjacency

One of the most common operations on graphs is asking two questions about adjacency: "Are these two items adjacent?" and more generally, "What items are adjacent to this item?" We can implement two `Graph` methods that answer these questions, using the same techniques as the previous section.

```
class Graph:
    def adjacent(self, item1: Any, item2: Any) -> bool:
        """Return whether item1 and item2 are adjacent vertices in this
         graph.

        Return False if item1 or item2 do not appear as vertices in this
         graph.
        """
        if item1 in self._vertices and item2 in self._vertices:
            v1 = self._vertices[item1]
            return any(v2.item == item2 for v2 in v1.neighbours)
        else:
            # We didn't find an existing vertex for both items.
            return False

    def get_neighbours(self, item: Any) -> set:
        """Return a set of the neighbours of the given item.

        Note that the *items* are returned, not the _Vertex objects
         themselves.

        Raise a ValueError if item does not appear as a vertex in this graph.
        """
        if item in self._vertices:
            v = self._vertices[item]
            return {neighbour.item for neighbour in v.neighbours}
        else:
            raise ValueError
```

And to wrap up, here's a doctest example putting together all of the methods we developed in this section:

```
>>> g = Graph()
>>> g.add_vertex('a')
```

```
>>> g.add_vertex('b')
>>> g.add_vertex('c')
>>> g.add_vertex('d')
>>> g.add_edge('a', 'b')
>>> g.add_edge('b', 'c')
>>> g.adjacent('a', 'b')
True
>>> g.adjacent('a', 'd')
False
>>> g.get_neighbours('b') == {'a', 'c'}
True
>>> g.get_neighbours('d') == set()
True
```