implementation of graph in python

Graphs are networks consisting of nodes connected by edges or arcs. In directed graphs, the connections between nodes have a direction, and are called arcs; in undirected graphs, the connections have no direction and are called edges. We mainly discuss directed graphs. Algorithms in graphs include finding a path between two nodes, finding the shortest path between two nodes, determining cycles in the graph (a cycle is a non-empty path from a node to itself), finding a path that reaches all nodes (the famous "traveling salesman problem"), and so on. Sometimes the nodes or arcs of a graph have weights or costs associated with them, and we are interested in finding the cheapest path.

There's considerable literature on graph algorithms, which are an important part of discrete mathematics. Graphs also have much practical use in computer algorithms. Obvious examples can be found in the management of networks, but examples abound in many other areas. For instance, caller-callee relationships in a computer program can be seen as a graph (where cycles indicate recursion, and unreachable nodes represent dead code).

Few programming languages provide direct support for graphs as a data type, and Python is no exception. However, graphs are easily built out of lists and dictionaries. For instance, here's a simple graph (I can't use drawings in these columns, so I write down the graph's arcs):

```
    A -> B

    A -> C

    B -> C

    B -> D

    C -> D

    D -> C

    E -> F

    F -> C
```

This graph has six nodes (A-F) and eight arcs. It can be represented by the following Python data structure:

```
    graph = {'A': ['B', 'C'],
             'B': ['C', 'D'],

             'C': ['D'],

             'D': ['C'],

             'E': ['F'],

             'F': ['C']}
```

This is a dictionary whose keys are the nodes of the graph. For each key, the corresponding value is a list containing the nodes that are connected by a direct arc from this node. This is about as simple as it gets (even simpler, the nodes could be represented by numbers instead of names, but names are more convenient and can easily be made to carry more information, such as city names).
Let's write a simple function to determine a path between two nodes. It takes a graph and the start and end nodes as arguments. It will return a list of nodes (including the start and end nodes) comprising the path. When no path can be found, it returns None. The same node will not occur more than once on the path returned (i.e. it won't contain cycles). The algorithm uses an important technique called *backtracking*: it tries each possibility in turn until it finds a solution.

```
    def find_path(graph, start, end, path=[]):

        path = path + [start]

        if start == end:

            return path

        if not graph.has_key(start):

            return None

        for node in graph[start]:

            if node not in path:

                newpath = find_path(graph, node, end, path)

                if newpath: return newpath

        return None
```

A sample run (using the graph above):

```
    >>> find_path(graph, 'A', 'D')
    ['A', 'B', 'C', 'D']

    >>>
```

The second 'if' statement is necessary only in case there are nodes that are listed as end points for arcs but that don't have outgoing arcs themselves, and aren't listed in the graph at all. Such nodes could also be contained in the graph, with an empty list of outgoing arcs, but sometimes it is more convenient not to require this.

Note that while the user calls `find_path()` with three arguments, it calls itself with a fourth argument: the path that has already been traversed. The default value for this argument is the empty list, '[]', meaning no nodes have been traversed yet. This argument is used to avoid cycles (the first 'if' inside the 'for' loop). The 'path' argument is not modified: the assignment "path = path + [start]" creates a new list. If we had written "path.append(start)" instead, we would have modified the variable 'path' in the caller, with disastrous results. (Using tuples, we could have been sure this would not happen, at the cost of having to write "path = path + (start,)" since "(start)" isn't a singleton tuple -- it is just a parenthesized expression.)

It is simple to change this function to return a list of all paths (without cycles) instead of the first path it finds:

```
    def find_all_paths(graph, start, end, path=[]):

        path = path + [start]

        if start == end:

            return [path]

        if not graph.has_key(start):

            return []

        paths = []

        for node in graph[start]:

            if node not in path:

                newpaths = find_all_paths(graph, node, end, path)

                for newpath in newpaths:

                    paths.append(newpath)

        return paths
```

A sample run:

```
    >>> find_all_paths(graph, 'A', 'D')

    [['A', 'B', 'C', 'D'], ['A', 'B', 'D'], ['A', 'C', 'D']]

    >>>
```

Another variant finds the shortest path:

```
    def find_shortest_path(graph, start, end, path=[]):

        path = path + [start]

        if start == end:

            return path

        if not graph.has_key(start):

            return None

        shortest = None

        for node in graph[start]:

            if node not in path:

                newpath = find_shortest_path(graph, node, end, path)

                if newpath:

                    if not shortest or len(newpath) < len(shortest):

                        shortest = newpath

        return shortest
```

Sample run:

```
    >>> find_shortest_path(graph, 'A', 'D')

    ['A', 'C', 'D']

    >>>
```