6.009: Fundamentals of Programming

Lecture 5:
Objects, Graphs, Trees

[Slides, code on course website]

*Suggestion:* download code templates from Calendar page, open your laptop, and work on the lecture problems as we do them!
Using dicts to represent graphs

```
graph = {'A': ['B', 'C'],
         'B': ['C', 'D'],
         'C': ['D'],
         'D': ['C'],
         'E': ['F'],
         'F': ['C']}
```
Building a new datatype: DirectedGraph

```
#~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
##  Directed Graphs
#~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

class DirectedGraph:
    """ Supports operations on directed graphs. """
    def __init__(self, edges):
        """ initialize directed graph, edges is list of (A,B) for each edge from node A to node B."""
        self.nodes = {}
        for e in edges:
            if len(e) != 2:  # add other checks too!
                raise ValueError("malformed edge: "+str(e))
            self.nodes.setdefault(e[0],[]).append(e[1])
        # ensure every node appears in nodes
        self.nodes.setdefault(e[1],[])
all_nodes, all_edges, edges_from_node

```python
def all_nodes(self):
    """ iterate over all nodes in the graph. """
    return iter(self.nodes)

def all_edges(self):
    """ iterate over all edges in the graph. """
    for node, neighbors in self.nodes.items():
        for neighbor in neighbors:
            yield (node, neighbor)

def edges_from_node(self, node):
    """ iterate over all edges from given node. """
    if node not in self.nodes:
        raise ValueError('unrecognized node: ' + node)
    for e in self.nodes[node]:
        yield (node, e)
```
graph.find_path(start, end)

def find_path(self, start, end):
    """ return path from start to end as a list.
    Uses depth-first search. """

    def extend(start, path):
        # extend current path
        path = path + [start]  # don't append
        if start == end:
            return path
        # see if we can reach end via neighbor
        for neighbor in self.nodes[start]:
            # avoid cycles by not including node more than once
            if neighbor not in path:
                # recursively find path after adding neighbor
                newpath = extend(neighbor, path)
                if newpath is not None:
                    return newpath

        # no paths found with given path as prefix
        return None

    if start not in self.nodes:
        raise ValueError('unrecognized node: ' + start)
    if end not in self.nodes:
        raise ValueError('unrecognized node: ' + end)
    return extend(start, [])
graph.find_path("A","D")

```python
graph = {
    'A': ['B', 'C'],
    'B': ['C', 'D'],
    'C': ['D'],
    'D': ['C'],
    'E': ['F'],
    'F': ['C']
}
```

```python
find_path("A","D",[])
find_path("B","D",["A")
find_path("C","D",["A","B")
find_path("D","D",["A","B","C")
return ["A","B","C","D"]
```
graph.find_path("A", "E")

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

```
find_path("A", "E", [])
find_path("B", "E", ["A"])  
find_path("C", "E", ["A", "B"])  
find_path("D", "E", ["A", "B", "C"])  
    return None  
find_path("D", "E", ["A", "B"])  
    return None
find_path("D", "E", ["A", "B"])
    return None
find_path("C", "E", ["A", "B", "D"])  
    return None
find_path("C", "E", ["A"])
    return None
find_path("D", "E", ["A", "C"])  
    return None
find_path("D", "E", ["A", "C"])
    return None
find_path("C", "E", ["A"])
    return None

Follows each path to its end:  
*Depth-first search*
def all_paths(self, start, end):
    """iterate over all paths from start to end.
    Uses depth-first search. """
    def extend(start, path):
        # extend current path
        path = path + [start]  # don't append
        if start == end:
            # reached destination, yield path we found
            yield path
        return
        # see if we can reach end via neighbor
        for neighbor in self.nodes[start]:
            # avoid cycles by not including node more than once
            if neighbor not in path:
                # recursively find paths after adding neighbor
                yield from extend(neighbor, path)
            if start not in self.nodes:
                raise ValueError('unrecognized node: ' + start)
        if end not in self.nodes:
            raise ValueError('unrecognized node: ' + start)
        return extend(start, [])
def find_shortest_path(self, start, end):
    ''' return shortest path from start to end as a list. '''
    # for better approach: Google "Dijkstra's algorithm"
    all_paths = list(self.all_paths(start, end))
    all_paths.sort(key=lambda x: len(x))
    return [] if len(all_paths)==0 else all_paths[0]

Tests all paths!
Can take exponential time...
Binary Search Tree

• Each tree node has
  – Key
  – Value
  – Left child
  – Right child

• Keys are kept in sorted order
  – Use binary search to find a key
  – Lookup, insertion, deletion: $O(\log N)$ on average

• Property: key in each node must be $\geq$ any key in left subtree and $\leq$ any key in right subtree
# Binary search tree and associated methods

```python
class BSTree:
    def __init__(self, key, val, left=None, right=None):
        self.key = key
        self.val = val
        self.left = left
        self.right = right
```

## Binary search tree with dict-like methods

Each node in the tree holds a key-value pair.
Inserting (key,val) in BSTree

```python
# insert new value into tree
def insert(self, key, val):
    node = self
    while True:
        if key == node.key:
            node.val = val
            return
        elif key < node.key:
            # key belongs in left subtree
            if node.left is None:
                node.left = BSTree(key, val)
                return
            node = node.left
        else:
            # key belongs in right subtree
            if node.right is None:
                node.right = BSTree(key, val)
                return
            node = node.right
```
Access Data in BSTree: in, get

# does tree have a particular key?
def __contains__(self, key):
    node = self
    while node is not None:
        if key == node.key: return True
        node = node.left if key < node.key \
        else node.right
    return False

# find key in tree, return associated value
def get(self, key, default = None):
    node = self
    while node is not None:
        if key == node.key:
            return node.val
        node = node.left if key < node.key \
        else node.right
    return default
BSTree in Action

tree = BSTree(22, "root")
tree.insert(14, "A")
tree.insert(33, "B")
tree.insert(2, "C")
tree.insert(17, "D")
tree.insert(27, "E")
tree.insert(45, "F")
tree.insert(47, "G")

print(17 in tree, 46 in tree)
⇒ True False

print(tree.get(45), tree.get(0,"Bingo!"))
⇒ F Bingo!
def print_tree(tree, prefix = ''):
    if tree is None:
        print('%sNone' % prefix)
    else:
        print('%s%s: %s' % (prefix, tree.key, tree.val))
        print_tree(tree.left, prefix + '  ')
        print_tree(tree.right, prefix + '  ')

22: root
14: A
  2: C
  None
  None
17: D
  None
  None
33: B
27: E
  None
  None
45: F
  None
47: G
  None
  None
Properties: look like instance variables, implemented by procedures

Example: graph.min == smallest key,val in tree
          == left-most child

    # example synthetic property
    def get_min(self):
        # min == left-most child in tree
        node = self
        while node.left: node = node.left
        # return key, value
        return (node.key, node.val)
    min = property(get_min)

In general: property(getter, setter, deleter, doc)
Decorators: @property

```python
@property
def min(self):
    # min == left-most child in tree
    node = self
    while node.left: node = node.left
    # return key, value
    return (node.key, node.val)
```

Easy to read, less clutter in method namespace
Iterating Over Keys in BSTree

```python
# iterator support
def __iter__(self):
    keys = []
    def walk(node):
        if node is None: return
        walk(node.left)
        keys.append(node.key)
        walk(node.right)
    walk(self)
    return iter(keys)
```

But this generates all the keys, even if user of iteration doesn’t want them all because the iteration was halted early.
A Better Iterator Implementation

```python
# a more incremental iterator
def __iter__(self):
    if self.left:
        for key in self.left:
            yield key
    yield self.key
    if self.right:
        for key in self.right:
            yield key
```

Generators are a simple and powerful tool for creating iterators. They are written like regular functions but use the `yield` statement whenever they want to return data. Each time `next()` is called on it, the generator resumes where it left off (it remembers all the data values and which statement was last executed).
A Even Better Iterator Implementation

```python
# a better incremental iterator
def __iter__(self):
    if self.left:
        yield from self.left
    yield self.key
    if self.right:
        yield from self.right

print(iter(tree))
⇒ <generator object BSTree.__iter__ at 0x10f124f10>

for key in tree: print(key)
⇒ 2
   14
   17 ...
   47
```
In-order access

# return in-order list of (key,value) pairs

def items(self):
    items = []
    def walk(node):
        if node is None: return
        walk(node.left)
        items.append((node.key, node.val))
        walk(node.right)
    walk(self)
    return items

print(tree.items())
⇒ [(2, 'C'), (14, 'A'), (17, 'D'), (22, 'root'),
   (27, 'E'), (33, 'B'), (45, 'F'), (47, 'G')]
In-order access (generator version)

```python
# generate in-order list of (key,value) pairs
def items(self):
    if self.left:
        yield from self.left.items()
    yield (self.key, self.val)
    if self.right:
        yield from self.right.items()

print(tree.items())
⇒ <generator object BSTree.items at 0x10f124f10>

print(list(tree.items()))
⇒ [(2, 'C'), (14, 'A'), (17, 'D'), (22, 'root'),
    (27, 'E'), (33, 'B'), (45, 'F'), (47, 'G')]
```
Making tree[key] work

Google “python special method names” ⇒ § 3.3 of Python3 Reference

```python
# support tree[key]
def __getitem__(self, key):
    if key not in self:
        raise KeyError
    else:
        return self.get(key)

# support tree[key] = val
def __setitem__(self, key, value):
    self.insert(key, value)
```

Testing it out:
```
tree[23] = 'skidoo'
print(tree[23])
⇒ skidoo
print(tree[42])
⇒ KeyError
```