Python Programming

Classes and Graph Traversal

Rathindra Nath Dutta

Senior Research Fellow
Advanced Computing & Microelectronics Unit
Indian Statistical Institute, Kolkata

August 24, 2023

Classes

- A means of bundling data and functions together
- Almost everything in Python is an object
  with its own properties and methods

Classes

- A means of bundling data and functions together
- Almost everything in Python is an object
  with its own properties and methods
- Class definition is a blueprint
  Objects are real instances having some values

Classes

- A means of bundling data and functions together
- Almost everything in Python is an object
  with its own properties and methods
- Class definition is a blueprint
  Objects are real instances having some values
- Use name of the class to construct new objects

```
class class_name:
    stmt_1
    ⋮
    stmt_n
...
obj = class_name() # instantiation with class constructor
```

Class Members

- A simple example of a class

```
class MyClass:
    pass # a filler statement having no effect
...
obj = MyClass() # instantiation
```

Class Members

- A simple example of a class
- Creating object members: simply assign value to a (new) variable

```
class MyClass:
    pass # a filler statement having no effect
...
obj = MyClass() # instantiation
obj.x = 123 # dot denotes membership
print(obj.x)
```

Class Members

- A simple example of a class
- Creating object members: simply assign value to a (new) variable
- Deleting object members: use the `del` operator

```
class MyClass:
    pass # a filler statement having no effect
...
obj = MyClass() # instantiation
obj.x = 123 # dot denotes membership
print(obj.x)
del obj.x
# print(obj.x) # error
```

Class Methods

- A simple example of a class with methods

```python
class MyClass:
    def foo(self): # self points to calling object
        print('you have called foo')
    def func1(self, x=0): # self is the first parameter
        print('you have called func1 with', x)
...
obj = MyClass() # instantiation
obj.foo() # calling member function
obj.func1()
obj.func1(123)
```

Class Methods

- A simple example of a class with methods
- Class methods can also access members created (later) outside

```
class MyClass:
    def foo(self): # self points to calling object
        print('you have called foo')
    def func1(self, x=0): # self is the first parameter
        print('you have called func1 with', x)
    def func2(self):
        print('you have called func2')
        print('value at a is', self.a) # accessing member
...
obj = MyClass() # instantiation
# obj.func2() # error
obj.a = 123
obj.func2()
```

More on Class

- All class members are public
  There is no concept of private member in Python

More on Class

- All class members are public
  There is no concept of private member in Python
- To denote a 'private' member in Python
  Prefix an identifier with underscore: `_x, __my_private_var`

More on Class

- All class members are public
  There is no concept of private member in Python
- To denote a 'private' member in Python
  Prefix an identifier with underscore: `_x, __my_private_var`
- For initializing class members at the time object creation

```
def __init__(self): # automatically invoked by constructor
    self.x = 123
```

More on Class

- All class members are public
  There is no concept of private member in Python
- To denote a 'private' member in Python
  Prefix an identifier with underscore: `_x`, `__my_private_var`
- For initializing class members at the time object creation

```
def __init__(self): # automatically invoked by constructor
    self.x = 123
```

- We may also pass values for object initialization

```
class Point:
    def __init__(self, x, y, z=0):
        self.x = x
        self.y = y
        self.z = z
...
p = Point(1, 2) # Use p.__dict__ to inspect member fields
p = Point(1, 2, 3) # Use p.__dir__() to list all members
```

Defining a Stack Class

- Field members:

Defining a Stack Class

- Field members: a `list` to store items, maybe a `top`/`size` variable

Defining a Stack Class

- Field members: a `list` to store items, maybe a `top`/`size` variable
- Member methods:

Defining a Stack Class

- Field members: a `list` to store items, maybe a `top`/`size` variable
- Member methods: initialization, `push()`, `pop()`, `size()`, `is_empty()`, `peek()`

Defining a Stack Class (contd.)

```python
class Stack:
    def __init__(self):
        self._arr = []
        self._size = 0

    def push(self, item):
        self._arr.append(item)
        self._size += 1

    def pop(self):
        if self._size == 0:
            return None # NULL in python
        self._size -= 1
        return self._arr.pop() # list method

    def size(self):
        return self._size

    def is_empty(self):
        return self._size == 0

    def peek(self): # only view
        if self._size == 0:
            return None
        return self._arr[-1]
```

Defining a Stack Class (contd.)

```
s = Stack()

s.push(10)
s.push(20)

print( s.size() )
print( s.pop() )
print( s.size() )

print( s.is_empty() )
print( s.peek() )
print( s.pop() )

print( s.pop() )

print( s.is_empty() )
```

- Implement a queue class

- Implement a queue class
- Check for balanced parenthesis (with operands and operators)

Class and Instance Variables

Example: object count

```
class Abc:
    count = 0 # shared class variable

    def __init__(self, x):
        self.x = x # object variable
        Abc.count += 1 # use class name to qualify
        print(Abc.count, self.x)

    Abc(10)
    Abc(20)
    obj = Abc(30)
    print(Abc.count)
    print(obj.count) # also possible
    print(obj.x)
    # print(Abc.x) # error
```

Graphs

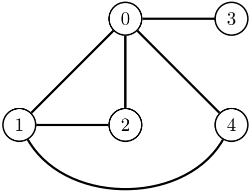
Nodes: model some real entity/state
Edges: encodes relationship among nodes

Graphs

Nodes: model some real entity/state
Edges: encodes relationship among nodes
There might be weights and other metadata as well

Graphs

Nodes: model some real entity/state
Edges: encodes relationship among nodes
There might be weights and other metadata as well

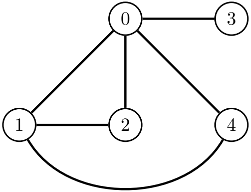


Graphs

Nodes: model some real entity/state

Edges: encodes relationship among nodes

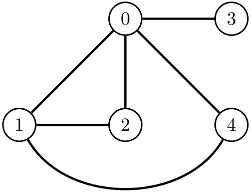
There might be weights and other metadata as well



	0	1	2	3	4
0	0	1	1	1	1
1	1	0	1	0	1
2	1	1	0	0	0
3	1	0	0	0	0
4	1	1	0	0	0

Graphs

Nodes: model some real entity/state
Edges: encodes relationship among nodes
There might be weights and other metadata as well



	0	1	2	3	4
0	0	1	1	1	1
1	1	0	1	0	1
2	1	1	0	0	0
3	1	0	0	0	0
4	1	1	0	0	0

0 : [1, 2, 3, 4]
1 : [0, 2, 4]
2 : [0, 1]
3 : [0]
4 : [0, 1]

Storing Graphs as Adjacency Lists

- Field members:

Storing Graphs as Adjacency Lists

- Field members: a list of lists to store node adjacency, maybe sizes

Storing Graphs as Adjacency Lists

- Field members: a list of lists to store node adjacency, maybe sizes
- Member methods:

Storing Graphs as Adjacency Lists

- Field members: a list of lists to store node adjacency, maybe sizes
- Member methods: initialization, `add_edge()`

Storing Graphs as Adjacency Lists

- Field members: a list of lists to store node adjacency, maybe sizes
- Member methods: initialization, `add_edge()`
  also maybe `is_adjacent()`, `get_neighbours()`, ...

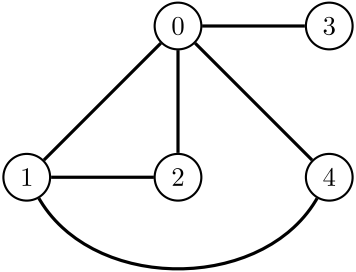
Storing Graphs as Adjacency Lists

- Field members: a list of lists to store node adjacency, maybe sizes
- Member methods: initialization, `add_edge()`
  also maybe `is_adjacent()`, `get_neighbours()`, . . .

```python
class Graph:
    def __init__(self, n):
        self._vertex_count = n
        self._adj_list = [ [] for _ in range(n) ]

    def add_edge(self, u, v):
        self._adj_list[u].append(v)
        self._adj_list[v].append(u) # undirected

    def is_adjacent(self, u, v):
        return v in self._adj_list[u]

    def get_neighbours(self, v):
        return self._adj_list[v]
```

Breadth First Search

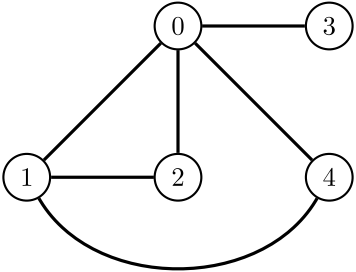
idea + algo



BFS starting at 0: 0, 1, 2, 3, 4
BFS starting at 1: 1, 0, 2, 4, 3

Breadth First Search

idea + algo

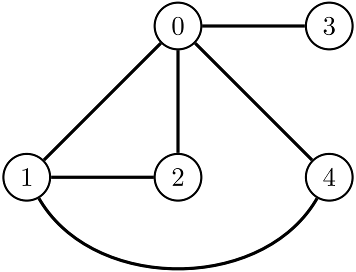


BFS starting at 0: 0, 1, 2, 3, 4
BFS starting at 1: 1, 0, 2, 4, 3

why queue, keeping track of visited nodes

Breadth First Search

idea + algo



BFS starting at 0: 0, 1, 2, 3, 4
BFS starting at 1: 1, 0, 2, 4, 3

why queue, keeping track of visited nodes

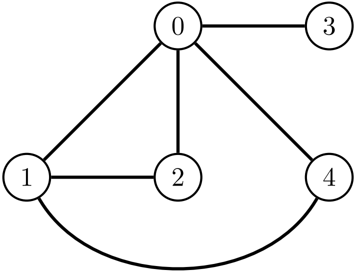
use of OPEN and CLOSED list

Procedure BFS()

```python
from my_queue import Queue # to import a Queue library
from graph import Graph # to import a Graph library
...
def BFS(g, source = 0):
    OPEN = Queue()
    CLOSED = []
    OPEN.enqueue(source)
    CLOSED.append(source) # visited
    while( not OPEN.is_empty() ):
        u = OPEN.dequeue()
        print(u, end=', ') # process node u
        neighbours = g.get_neighbours(u)
        for v in neighbours:
            if v not in CLOSED: # not yet visited
                OPEN.enqueue(v)
                CLOSED.append(v) # visited
    print() # print a newline
...
BFS(g)
BFS(g, 1)
```

Depth First Search

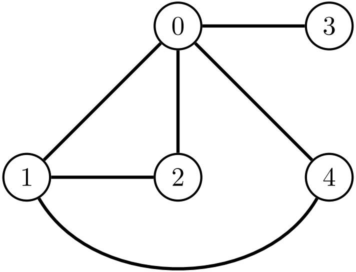
idea + algo



DFS starting at 0: 0, 1, 2, 4, 3
DFS starting at 2: 2, 0, 1, 4, 3

Depth First Search

idea + algo

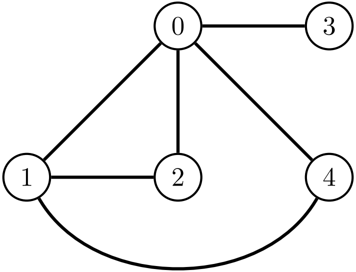


DFS starting at 0: 0, 1, 2, 4, 3
DFS starting at 2: 2, 0, 1, 4, 3

recursion

Depth First Search

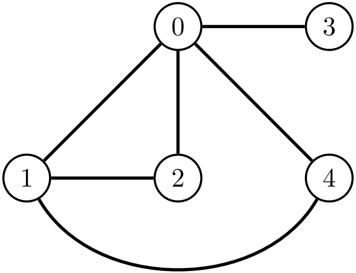
idea + algo



DFS starting at 0: 0, 1, 2, 4, 3
DFS starting at 2: 2, 0, 1, 4, 3
recursion: CLOSED must be global/or passed as an argument

Depth First Search

idea + algo



DFS starting at 0: 0, 1, 2, 4, 3

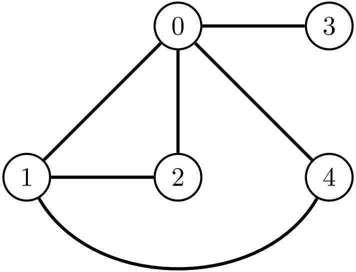
DFS starting at 2: 2, 0, 1, 4, 3

recursion: CLOSED must be global/or passed as an argument

pros & cons of recursive methods

Depth First Search

idea + algo



DFS starting at 0: 0, 1, 2, 4, 3

DFS starting at 2: 2, 0, 1, 4, 3

recursion: CLOSED must be global/or passed as an argument

pros & cons of recursive methods

why stack

Procedure DFS()

Recursive implementation

```
from graph import Graph # to import a Graph library
...
def DFS(g, source=0, CLOSED=[]):
    CLOSED.append(source) # mark as current source as visited
    print(source, end=', ') # process current source
    neighbors = g.get_neighbours(source)
    for v in neighbors:
        if v not in CLOSED:
            DFS(g, v, CLOSED) # same CLOSED list reference passed
...
# DFS(g)
DFS(g, 2)
```

Procedure DFS()

Recursive implementation

```python
from graph import Graph # to import a Graph library
...
def DFS(g, source=0, CLOSED=[]):
    CLOSED.append(source) # mark as current source as visited
    print(source, end=', ') # process current source
    neighbors = g.get_neighbours(source)
    for v in neighbors:
        if v not in CLOSED:
            DFS(g, v, CLOSED) # same CLOSED list reference passed
...
# DFS(g)
DFS(g, 2)
```

Exercise: how would you print a newline at the end?

- Implement the following non-recursive version of DFS

```
procedure DFS(G, source):
    create a stack S
    S.push(source)
    while S is not empty do
        u = S.pop()
        if u is not yet visited then
            process/print node u
            mark source as visited
            forall neighbor v of u do
                if v is not yet visited then
                    mark v as visited
                    S.push(v)
```

- Implement the following non-recursive version of DFS

```
procedure DFS(G, source):
    create a stack S
    S.push(source)
    while S is not empty do
        u = S.pop()
        if u is not yet visited then
            process/print node u
            mark source as visited
            forall neighbor v of u do
                if v is not yet visited then
                    mark v as visited
                    S.push(v)
```

- Compare the output of recursive version and non-recursive version