

NPTEL MOOC

PROGRAMMING, DATA STRUCTURES AND ALGORITHMS IN PYTHON

Week 6, Lecture 4

Madhavan Mukund, Chennai Mathematical Institute

<http://www.cmi.ac.in/~madhavan>

Data structures

- * *Algorithms + Data Structures = Programs*
Niklaus Wirth
- * Arrays/lists — sequences of values
- * Dictionaries — key-value pairs
- * Python also has sets as a built in datatype

Sets in Python

- * List with braces, duplicates automatically removed

```
colours = {'red', 'black', 'red', 'green'}
```

```
>>> print(colours)
{'black', 'red', 'green'}
```

- * Create an empty set

```
colours = set()
```

- * Note, not `colours = {}` — empty dictionary!

Sets in Python

- * Set membership

```
>>> 'black' in colours  
True
```

- * Convert a list into a set

```
>>> numbers = set([0,1,3,2,1,4])  
>>> print(numbers)  
{0, 1, 2, 3, 4}
```

```
>>> letters = set('banana')  
>>> print(letters)  
{ 'a', 'n', 'b' }
```

Set operations

```
odd = set([1,3,5,7,9,11])  
prime = set([2,3,5,7,11])
```

- * Union

```
odd | prime → {1, 2, 3, 5, 7, 9, 11}
```

- * Intersection

```
odd & prime → {3, 11, 5, 7}
```

- * Set difference

```
odd - prime → {1, 9}
```

- * Exclusive or

```
odd ^ prime → {1, 2, 9}
```

Stacks

- * Stack is a last-in, first-out list
 - * `push(s,x)` — add `x` to stack `s`
 - * `pop(s)` — return most recently added element
- * Maintain stack as list, push and pop from the right
 - * `push(s,x)` is `s.append(x)`
 - * `s.pop()` — Python built-in, returns last element

Stacks

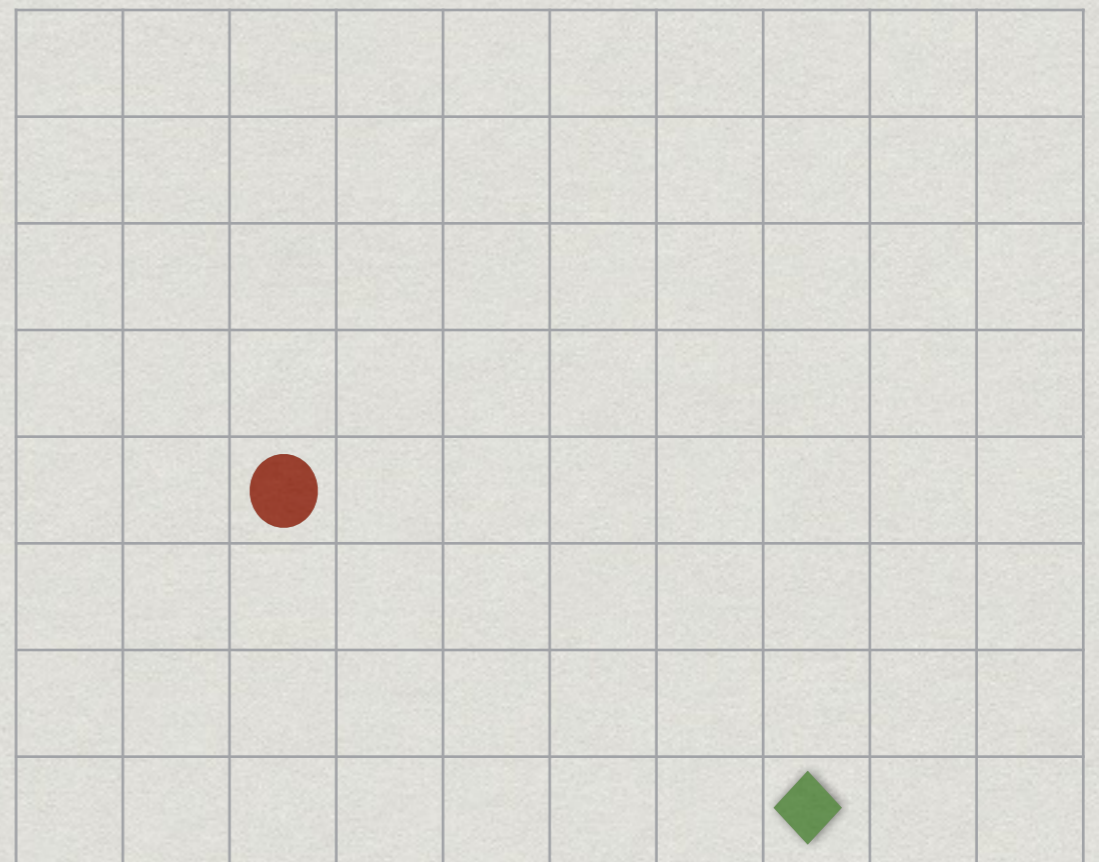
- * Stacks are natural to keep track of recursive function calls
- * In 8 queens, use a stack to keep track of queens added
 - * Push the latest queen onto the stack
 - * To backtrack, pop the last queen added

Queues


- * First-in, first-out sequences
 - * `addq(q, x)` — adds `x` to rear of queue `q`
 - * `removeq(q)` — removes element at head of `q`
- * Using Python lists, left is rear, right is front
 - * `addq(q, x)` is `q.insert(0, x)`
 - * `l.insert(j, x)`, insert `x` before position `j`
 - * `removeq(q)` is `q.pop()`

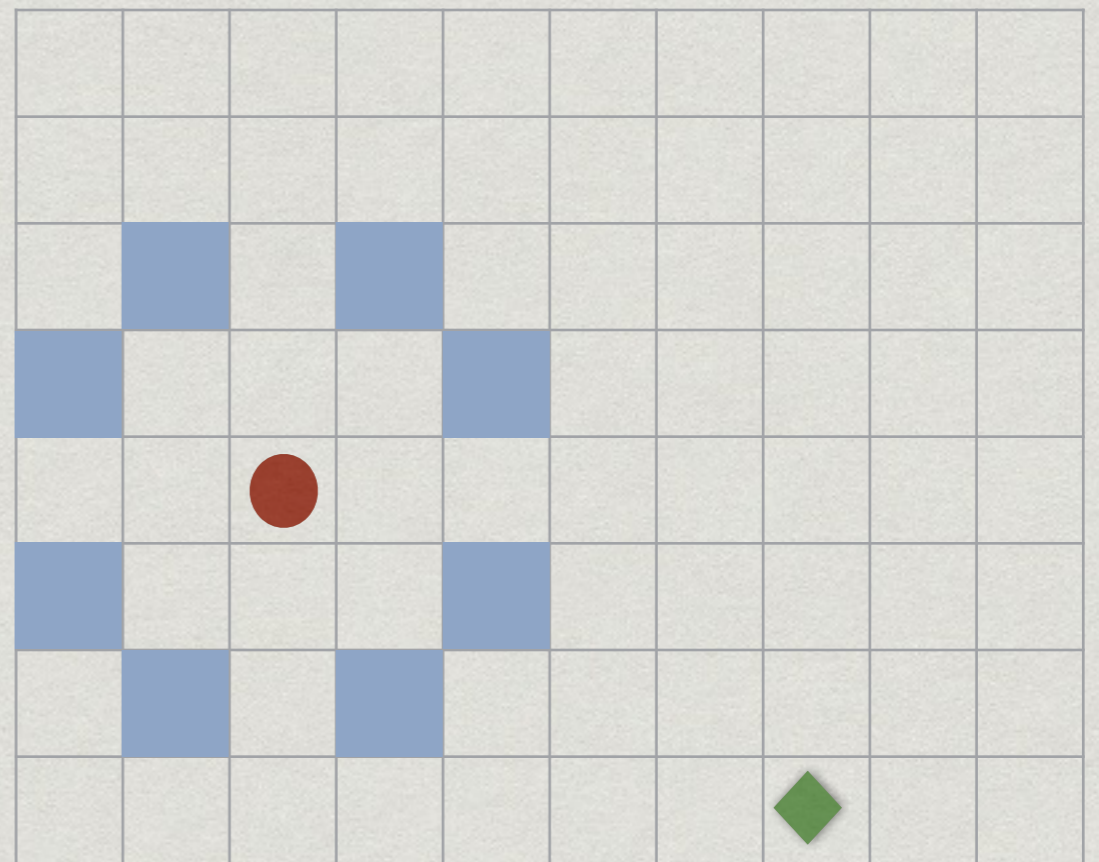
Systematic exploration

- * Rectangular $m \times n$ grid
- * Chess knight starts at (sx, sy)
 - * Usual knight moves
- * Can it reach a target square (tx, ty) ? ◆



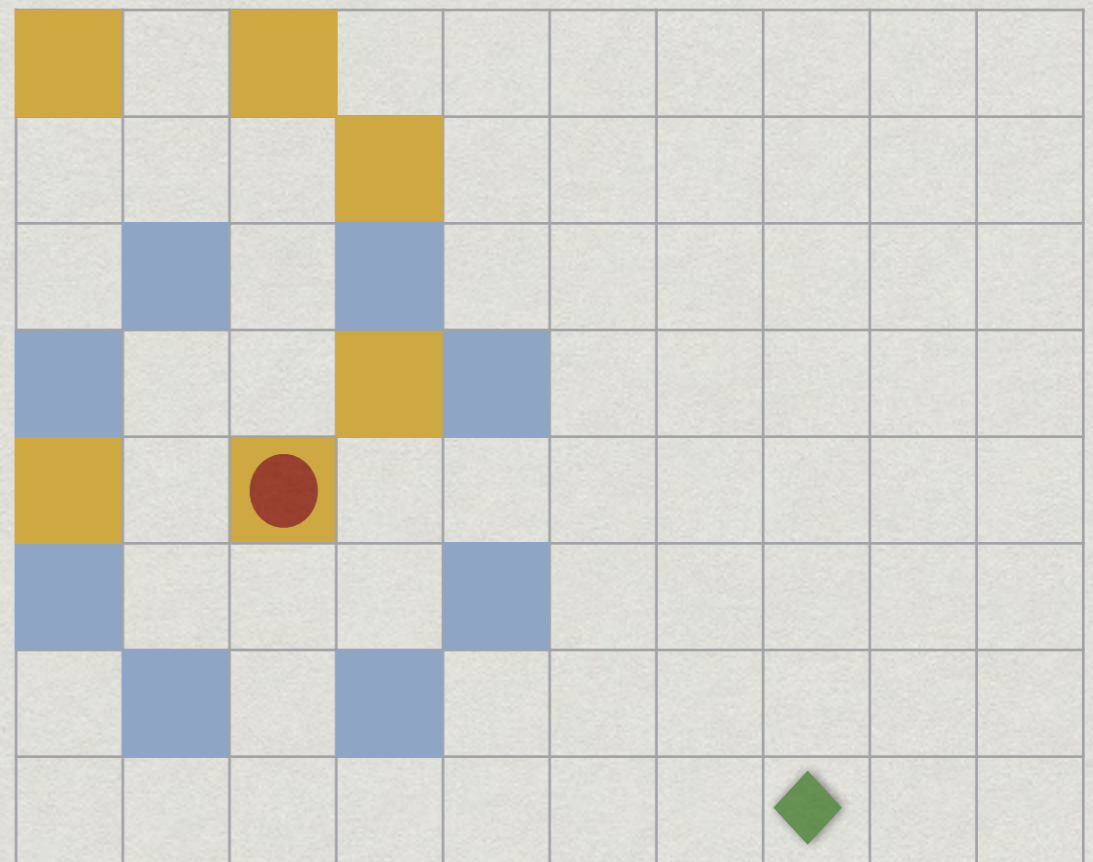
Systematic exploration

- * Rectangular $m \times n$ grid
- * Chess knight starts at (sx, sy)
- * Usual knight moves
- * Can it reach a target square (tx, ty) ? 




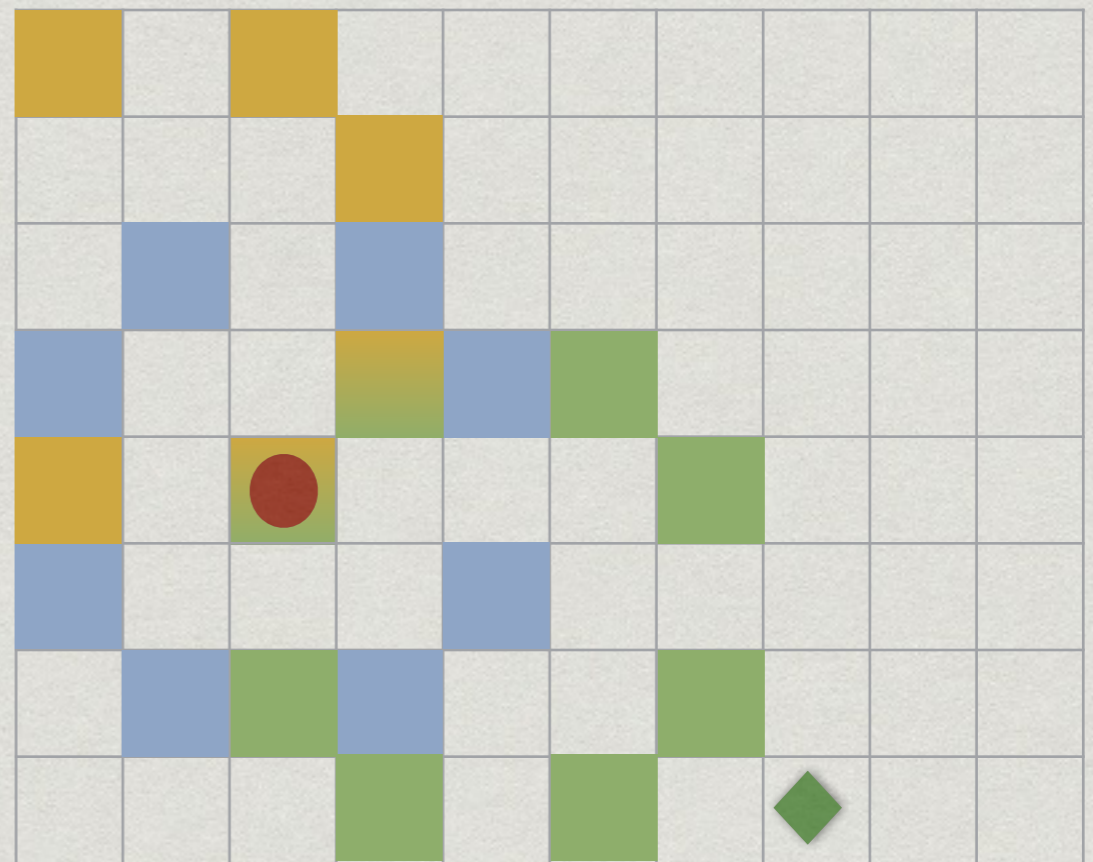
Systematic exploration

- * Rectangular $m \times n$ grid
- * Chess knight starts at (sx, sy)
- * Usual knight moves
- * Can it reach a target square (tx, ty) ? ◆



Systematic exploration

- * Rectangular $m \times n$ grid
- * Chess knight starts at (sx, sy)
- * Usual knight moves
- * Can it reach a target square (tx, ty) ? 



Systematic exploration

- * X1 — all squares reachable in one move from (sx,sy)
- * X2 — all squares reachable from X1 in one move
- * . . .
- * Don't explore an already marked square
- * When do we stop?
 - * If we reach target square
 - * What if target is not reachable?

Systematic exploration

- * Maintain a queue Q of cells to be explored
- * Initially Q contains only start node (sx, sy)
 - * Remove (ax, ay) from head of queue
 - * Mark all squares reachable in one step from (ax, ay)
 - * Add all newly marked squares to the queue
- * When the queue is empty, we have finished

Systematic exploration

```
def explore((sx,sy),(tx,ty)):
    marked = [[0 for i in range(n)]
               for j in range(m)]
    marked[sx][sy] = 1
    queue = [(sx,sy)]
    while queue != []:
        (ax,ay) = queue.pop()
        for (nx,ny) in neighbours((ax,ay)):
            if !marked[nx][ny]:
                marked[nx][ny] = 1
                queue.insert(0,(nx,ny))
    return(marked[tx][ty])
```

Example

src = (0,1) **tgt = (1,1)**



- * This is an example of **breadth first search**

Summary

- * Data structures are ways of organising information that allow efficient processing in certain contexts
- * Python has a built-in implementation of sets
- * Stacks are useful to keep track of recursive computations
- * Queues are useful for breadth-first exploration