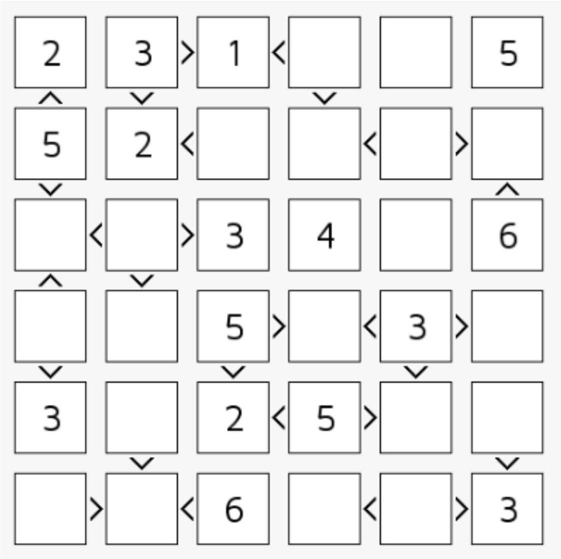
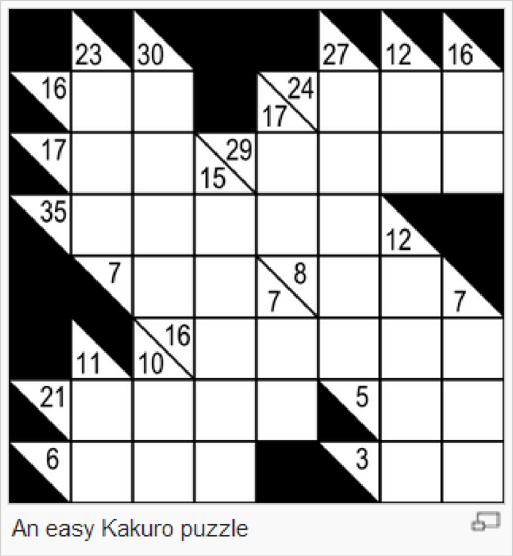
Object Oriented Programming 31695

Practice Problems





An easy Kakuro puzzle

Straight-row Sudoku Puzzles

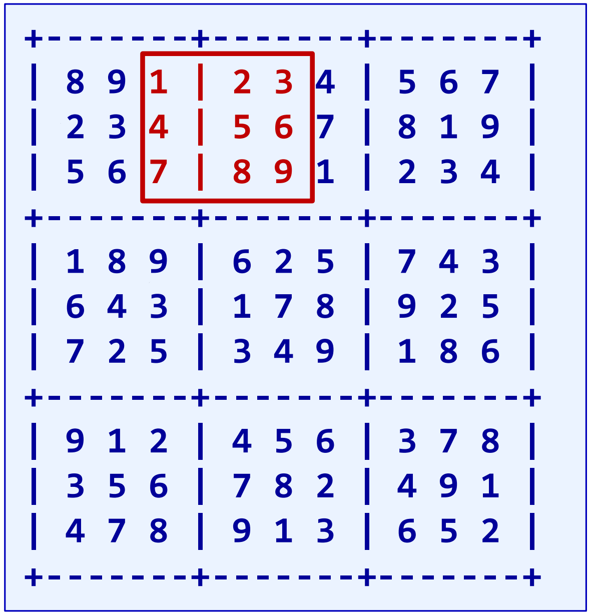
- **Definition:** A **straight-row Sudoku board** is a full 9x9 board that contains a valid Sudoku solution with a straight-row (numbers in natural order)
- **Straight-column Sudoku board** is defined similarly
- Use your Sudoku class to write a Python program for calculating how many straight-row Sudoku boards are there? Would inheritance be a more efficient way to solve the problem?
- Here are two examples of such boards:

```
+-------+-------+-------+
| 9 6 7 | 8 2 3 | 4 5 1 |
| 8 4 5 | 1 9 7 | 2 3 6 |
| 1 2 3 | 4 5 6 | 7 8 9 |
+-------+-------+-------+
| 2 9 8 | 5 6 1 | 3 7 4 |
| 6 3 1 | 2 7 4 | 8 9 5 |
| 7 5 4 | 9 3 8 | 1 6 2 |
+-------+-------+-------+
| 3 7 9 | 6 4 2 | 5 1 8 |
| 4 8 6 | 7 1 5 | 9 2 3 |
| 5 1 2 | 3 8 9 | 6 4 7 |
+-------+-------+-------+
```

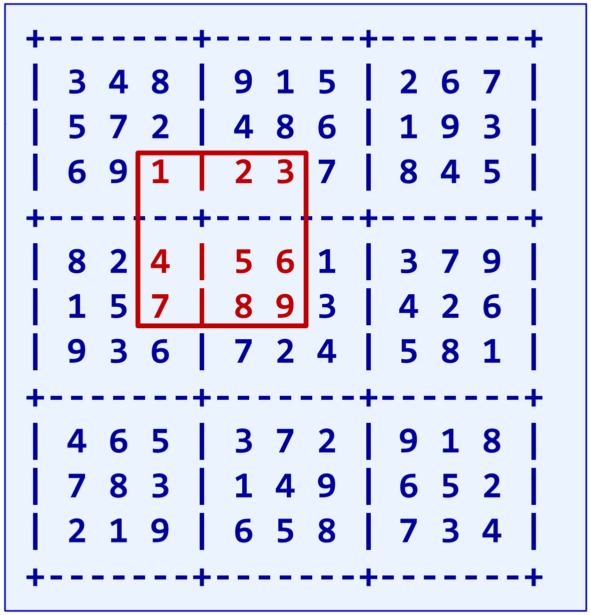
```
+-------+-------+-------+
| 2 9 8 | 5 6 1 | 3 7 4 |
| 6 3 1 | 2 7 4 | 8 9 5 |
| 7 5 4 | 9 3 8 | 1 6 2 |
+-------+-------+-------+
| 3 7 2 | 6 4 5 | 9 1 8 |
| 4 1 6 | 3 8 9 | 5 2 7 |
| 5 8 9 | 7 1 2 | 6 4 3 |
+-------+-------+-------+
| 8 4 5 | 1 9 7 | 2 3 6 |
| 1 2 3 | 4 5 6 | 7 8 9 |
| 9 6 7 | 8 2 3 | 4 5 1 |
+-------+-------+-------+
```

Straight-Block Sudoku Puzzles

- **Definition:** A **straight-block Sudoku board** is a full 9x9 board that contains a valid Sudoku solution with a **straight-block** (3x3 sub-block with numbers in natural order – see below)
- Use your Sudoku class to write a Python program for calculating how many straight-block Sudoku boards are there?
- Here are two examples of such boards:



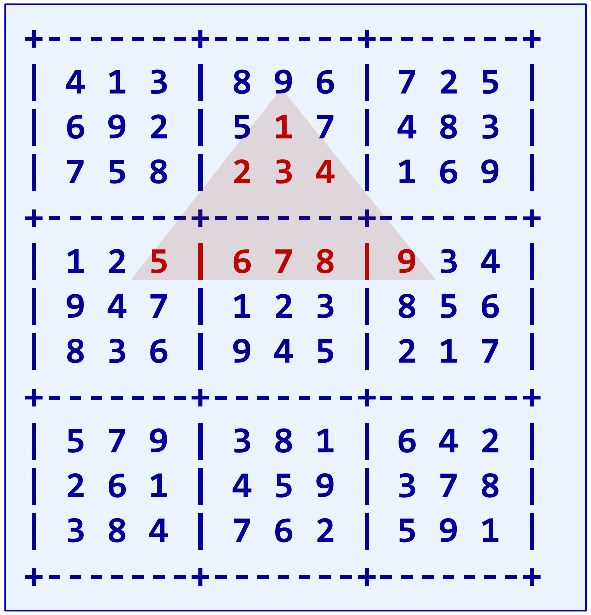
```
+-------+-------+-------+
| 8 9 1 | 2 3 4 | 5 6 7 |
| 2 3 4 | 5 6 7 | 8 1 9 |
| 5 6 7 | 8 9 1 | 2 3 4 |
+-------+-------+-------+
| 1 8 9 | 6 2 5 | 7 4 3 |
| 6 4 3 | 1 7 8 | 9 2 5 |
| 7 2 5 | 3 4 9 | 1 8 6 |
+-------+-------+-------+
| 9 1 2 | 4 5 6 | 3 7 8 |
| 3 5 6 | 7 8 2 | 4 9 1 |
| 4 7 8 | 9 1 3 | 6 5 2 |
+-------+-------+-------+
```



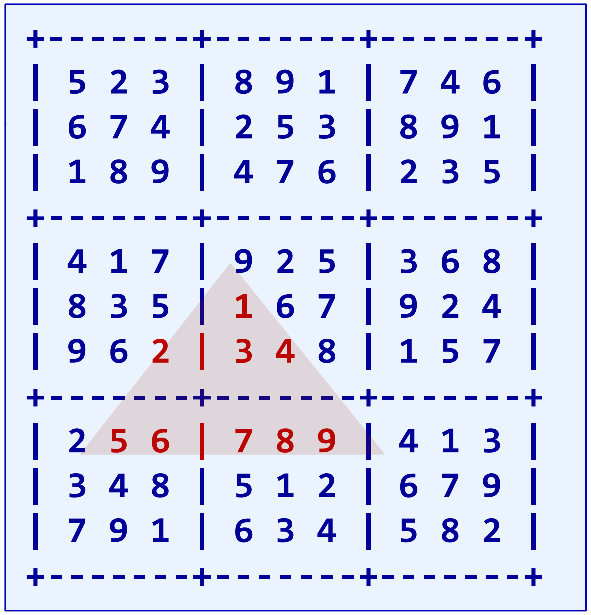
```
+-------+-------+-------+
| 3 4 8 | 9 1 5 | 2 6 7 |
| 5 7 2 | 4 8 6 | 1 9 3 |
| 6 9 1 | 2 3 7 | 8 4 5 |
+-------+-------+-------+
| 8 2 4 | 5 6 1 | 3 7 9 |
| 1 5 7 | 8 9 3 | 4 2 6 |
| 9 3 6 | 7 2 4 | 5 8 1 |
+-------+-------+-------+
| 4 6 5 | 3 7 2 | 9 1 8 |
| 7 8 3 | 1 4 9 | 6 5 2 |
| 2 1 9 | 6 5 8 | 7 3 4 |
+-------+-------+-------+
```

Straight-Triangle Sudoku Puzzles

- **Definition:** A **straight-triangle Sudoku board** is a full 9x9 board that contains a valid Sudoku solution with a **straight-triangle** (see examples below)
- Use your Sudoku class to write a Python program for calculating how many straight-triangle Sudoku boards are there?
- Here are two examples of such boards:



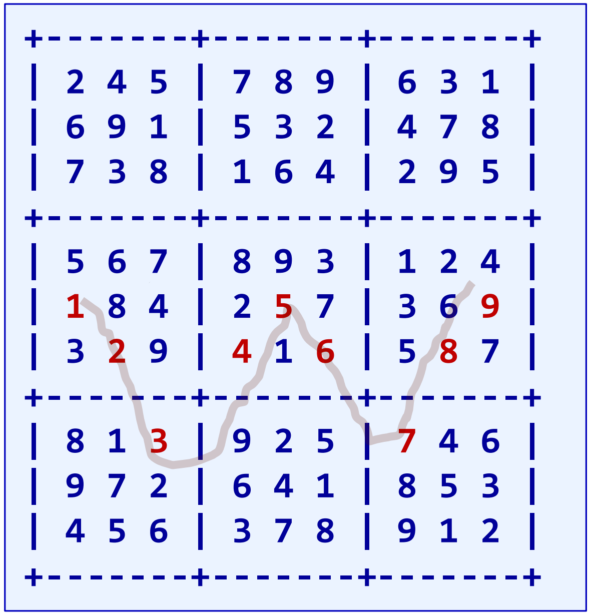
```
+-------+-------+-------+
| 4 1 3 | 8 9 6 | 7 2 5 |
| 6 9 2 | 5 1 7 | 4 8 3 |
| 7 5 8 | 2 3 4 | 1 6 9 |
+-------+-------+-------+
| 1 2 5 | 6 7 8 | 9 3 4 |
| 9 4 7 | 1 2 3 | 8 5 6 |
| 8 3 6 | 9 4 5 | 2 1 7 |
+-------+-------+-------+
| 5 7 9 | 3 8 1 | 6 4 2 |
| 2 6 1 | 4 5 9 | 3 7 8 |
| 3 8 4 | 7 6 2 | 5 9 1 |
+-------+-------+-------+
```



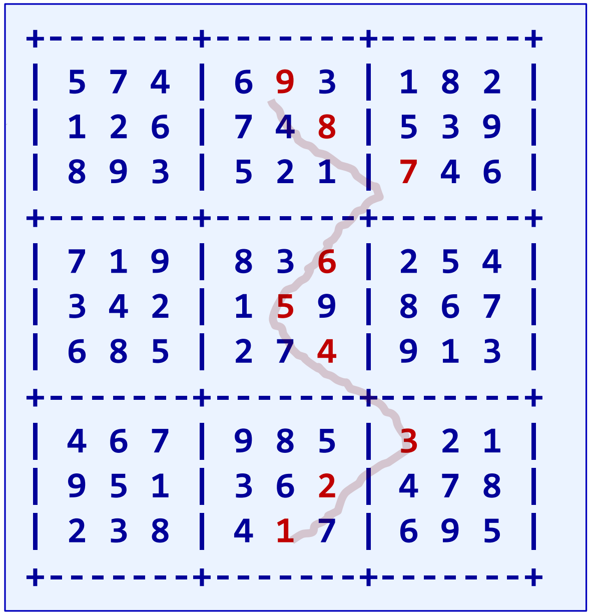
```
+-------+-------+-------+
| 5 2 3 | 8 9 1 | 7 4 6 |
| 6 7 4 | 2 5 3 | 8 9 1 |
| 1 8 9 | 4 7 6 | 2 3 5 |
+-------+-------+-------+
| 4 1 7 | 9 2 5 | 3 6 8 |
| 8 3 5 | 1 6 7 | 9 2 4 |
| 9 6 2 | 3 4 8 | 1 5 7 |
+-------+-------+-------+
| 2 5 6 | 7 8 9 | 4 1 3 |
| 3 4 8 | 5 1 2 | 6 7 9 |
| 7 9 1 | 6 3 4 | 5 8 2 |
+-------+-------+-------+
```

Straight-Wave Sudoku Puzzles

- **Definition:** A **straight-wave Sudoku board** is a full 9x9 board that contains a valid Sudoku solution with a **straight-wave pattern** (see examples below)
- Use your Sudoku class to write a Python program for calculating how many straight-wave Sudoku boards are there?
- Here are two examples of such boards:



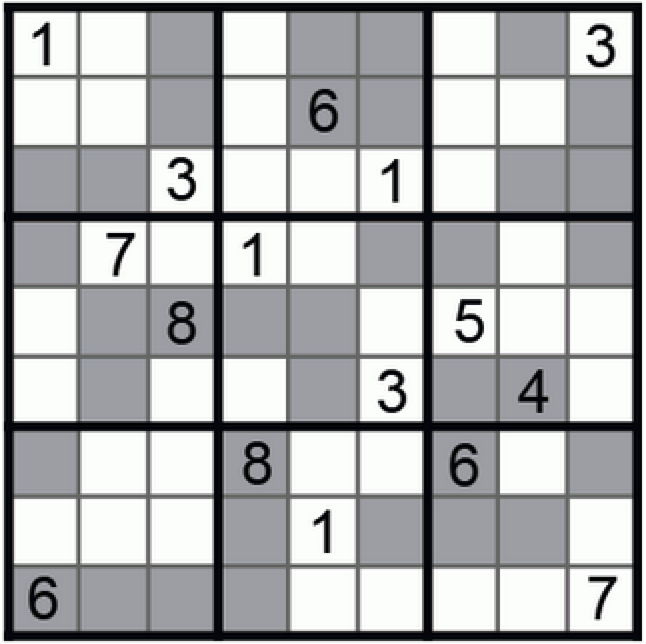
```
+-------+-------+-------+
| 2 4 5 | 7 8 9 | 6 3 1 |
| 6 9 1 | 5 3 2 | 4 7 8 |
| 7 3 8 | 1 6 4 | 2 9 5 |
+-------+-------+-------+
| 5 6 7 | 8 9 3 | 1 2 4 |
| 1 8 4 | 2 5 7 | 3 6 9 |
| 3 2 9 | 4 1 6 | 5 8 7 |
+-------+-------+-------+
| 8 1 3 | 9 2 5 | 7 4 6 |
| 9 7 2 | 6 4 1 | 8 5 3 |
| 4 5 6 | 3 7 8 | 9 1 2 |
+-------+-------+-------+
```



```
+-------+-------+-------+
| 5 7 4 | 6 9 3 | 1 8 2 |
| 1 2 6 | 7 4 8 | 5 3 9 |
| 8 9 3 | 5 2 1 | 7 4 6 |
+-------+-------+-------+
| 7 1 9 | 8 3 6 | 2 5 4 |
| 3 4 2 | 1 5 9 | 8 6 7 |
| 6 8 5 | 2 7 4 | 9 1 3 |
+-------+-------+-------+
| 4 6 7 | 9 8 5 | 3 2 1 |
| 9 5 1 | 3 6 2 | 4 7 8 |
| 2 3 8 | 4 1 7 | 6 9 5 |
+-------+-------+-------+
```

Even-Odd Sudoku

- Fill in the grid so that every row, column, 3x3 box, contains the digits 1 through 9
- Gray cells are even, white cells are odd
- Use your Sudoku class (by inheritance) to build an EvenOddSudoku class which solves this type of puzzles. Your class will be initialized by a board and a list of gray cells.
- Which methods you need to override? Write an ADT first



1		▨		▨	▨		▨	3
		▨		▨ 6	▨			▨
▨	▨	3			1		▨	▨
▨	7		1		▨	▨		▨
	▨	▨ 8	▨	▨		5		
	▨			▨	3	▨	▨ 4	
▨			▨ 8			▨ 6		▨
			▨	1	▨	▨	▨	
▨ 6	▨	▨	▨					7

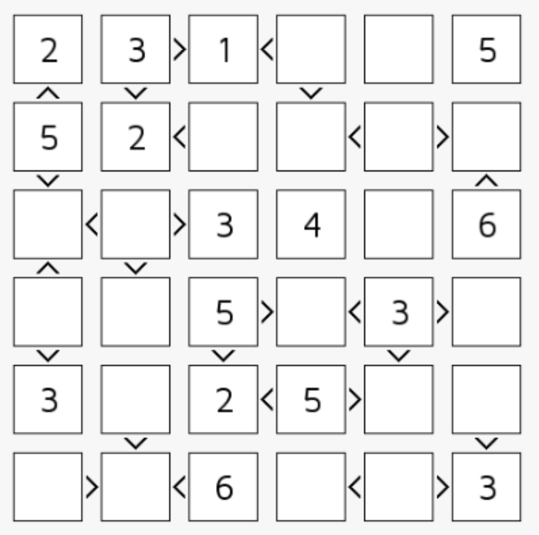
SUDOKU 6x6

- How hard is it to redesign a class for 6x6 Sudoku?
- ADT?
- Class skeleton
- Simple test

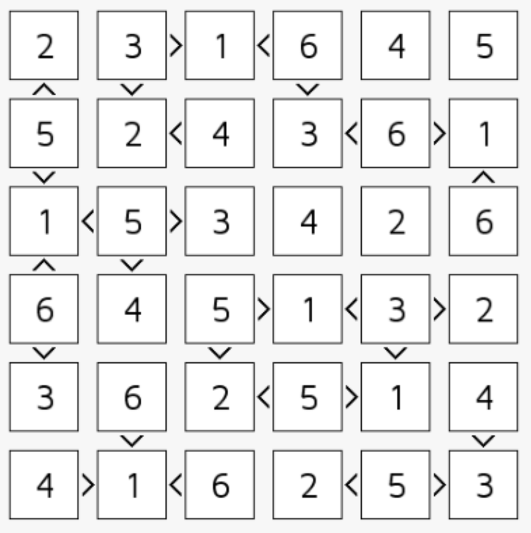
3		1	4		
4	5	6			3
				5	4
5	6	4	2	3	
	4		3	1	2

Futoshiki

- http://en.wikipedia.org/wiki/Futoshiki
- Each row and each columns must contain all six digits, but also must honor inequality signs (“constraints”)
- Suggest an ADT (Abstract Data Type) for a **Futoshiki Solver**
- Write a simple test for that solver. It should test that the solver works for at least one puzzle
- Board size can vary! 9x9, 12x12, 16x16, …



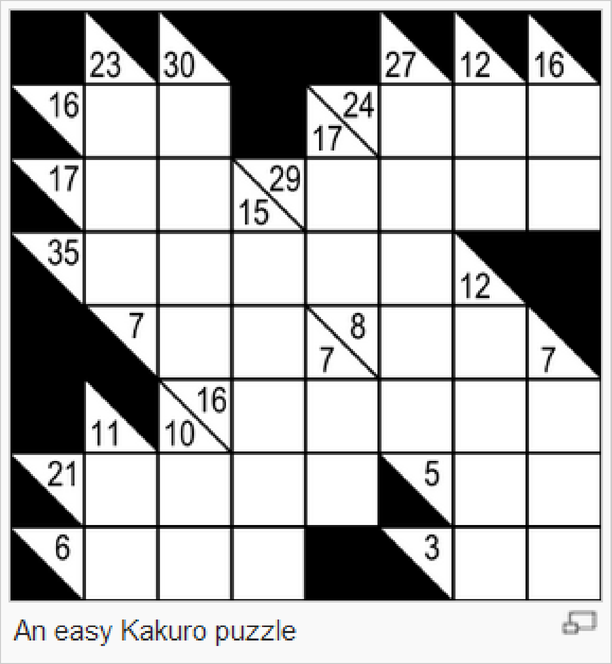
```
2   3 > 1 <         5
^   v       v
5   2 <       <   >
v                   ^
  <   > 3   4       6
^   v
        5 >   < 3 >
v       v       v
3       2 < 5 >
    v               v
  >   < 6     <   > 3
```



```
2   3 > 1 < 6   4   5
^   v       v
5   2 < 4   3 < 6 > 1
v                   ^
1 < 5 > 3   4   2   6
^   v
6   4   5 > 1 < 3 > 2
v       v       v
3   6   2 < 5 > 1   4
    v               v
4 > 1 < 6   2 < 5 > 3
```

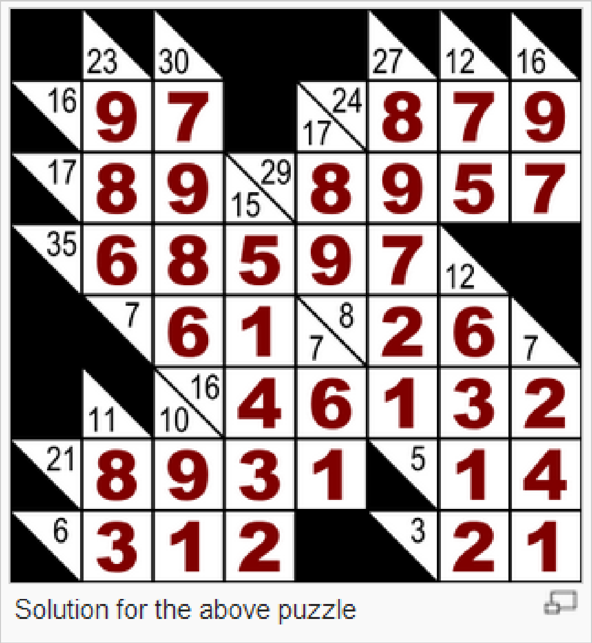
Kakuro

- http://en.wikipedia.org/wiki/Kakuro
- Like an ordinary crossword puzzle but with numbers and sums
- Easy to understand from the following example
- Boards can be of any size! 5x5, 8x8, 12x12, 16x16, etc



■	23\	30\	■	■	27\	12\	16\
\16			■	17\24			
\17			15\29				
\35						12\	■
■	\7			7\8			7\
■	11\	10\16					
\21					\5		
\6				■	\3		

An easy Kakuro puzzle

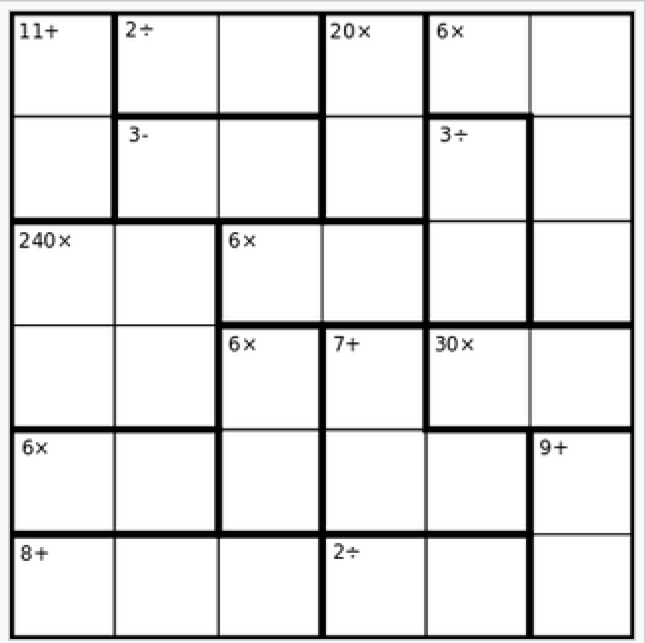


■	23\	30\	■	■	27\	12\	16\
\16	9	7	■	17\24	8	7	9
\17	8	9	15\29	8	9	5	7
\35	6	8	5	9	7	12\	■
■	\7	6	1	7\8	2	6	7\
■	11\	10\16	4	6	1	3	2
\21	8	9	3	1	\5	1	4
\6	3	1	2	■	\3	2	1

Solution for the above puzzle

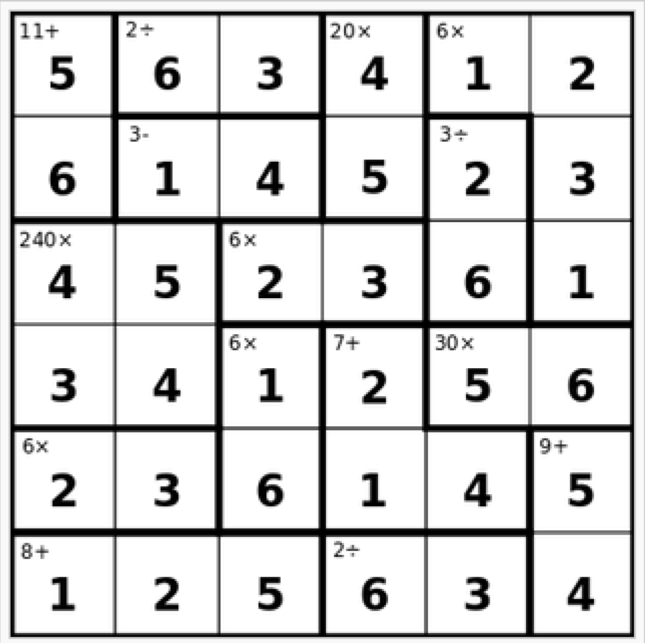
Kendoku (Kenken)

- http://en.wikipedia.org/wiki/KenKen
- invented in 2004 by Japanese math teacher Tetsuya Miyamoto
- who intended the puzzles to be an instruction-free method of training the brain
- Board size varies: 6x6, 8x8, 12x12, 16x16, etc.



11+	2÷		20×	6×	
	3-			3÷	
240×		6×			
		6×	7+	30×	
6×					9+
8+			2÷		

A typical KenKen problem.



11+ 5	2÷ 6	3	20× 4	6× 1	2
6	3- 1	4	5	3÷ 2	3
240× 4	5	6× 2	3	6	1
3	4	6× 1	7+ 2	30× 5	6
6× 2	3	6	1	4	9+ 5
8+ 1	2	5	2÷ 6	3	4

Solution to the above problem.

Bounded Stack

- Look at the Stack ADT we did in class
- Add a new requirement: the Stack length must be limited by a given size MAXLEN
- Implement this new BoundedStack class
- Can it be done by inheritance from Stack?
- Use a FullStack Exception class in your implementation

Bounded Stack ADT

- s = BundedStack(maxsize) **Constructor**

  Create a new BoundedStack object with maximal size = maxsize

- s.push(item) **Mutator**

  push a new item to the BoundedStack

  make sure stack size does not exceed maxsize

- s.pop() **Mutator**

  pop an item from the stack

  raise an exception if stack is empty (EmptyStack)

- s.peek() Accessor

  return head of stack

- s.is_empty() Accessor
- s.size() Acccessor

Efficient Queue List Implementation

- The two list implementations we saw in class had an O(n) complexity in one of the methods: enqueue, dequeue
- Use two self.tail and self.head members to fix this problem
- Make sure that list memory is constrained

Using Stack and Queue

- Show how to use a stack *s* and a Queue *q* to generate all possible subsets of an *n*-element set *T* non-recursively
- Write an iterator class based on this idea

- Describe how to implement the stack ADT using a single queue as a data member, and only constant additional local memory within the method bodies
- What is the running time of the push(), pop(), and peek() methods for your design?

- Describe how to implement the queue ADT using two stacks as data members, such that all queue operations execute in amortized *O*(1) time.

Using Stack and Queue

- Describe how to implement the double-ended queue ADT using two stacks as data members
- What are the running times of the methods?
- Suppose you have a stack *s* containing *n* elements and a queue *q* that is initially empty. Describe how you can use *q* to scan *s* to see if it contains a certain element *x*, with the additional constraint that your algorithm must return the elements back to *s* in their original order
- You may only use *s*, *q*, and a constant number of other variables

Bounded Queue

- Look at the Queue ADT we did in class
- Add a new requirement: the Queue length must be limited by a given size MAXLEN
- Implement this new BoundedQueue class
- Can it be done by inheritance from Queue?
- Use a FullQueue Exception class in your implementation

Linked List

- Implement LinkedList class based on our **Node** class

```
class LinkedList:
    def __init__(self):
        self.first = None
        self.last = None

    def insert(self, item):    # Time complexity = O(1)
        pass

    def remove(self, item):    # Time complexity = ?
        pass                   # Left as an exercise!

    def reverse(self):         # Return a reversed linked list
        pass                   # Left as an exercise

    def index(self, item):     # return first index of data in list
        pass                   # Left as an exercise. Complexity = ?

    def __str__(self):
        pass
```

The Link Class

- To define a doubly linked list, we will need a new type of link element

```
class Link(object):
    def __init__(self, data, prev=None, next=None):
        self.data = data
        self.prev = prev
        self.next = next

    def __str__(self):
        return 'Link(%s)' % str(self.data)
```

Testing the Link Class

- Explain what the following test does?

```
def test1():
    a = Link('Alice')
    b = Link('Bob', a)
    c = Link('Clod', b)
    d = Link('Dian', c)
    e = Link('Eddi', d)
    a.next = b
    b.next = c
    c.next = d
    d.next = e

    assert a.next.prev is a
    assert e.prev.prev is c
    assert a.next.next.next is d
    assert e.data == 'Eddi'
    assert d.data == 'Dian'
    assert a.prev is None
    print "test1 PASSED"
```

Getting the following links

- Write a function **forward_links(x)** which lists all the links that follow x

```
def test3():
    a = Link('Alice')
    b = Link('Bob', a)
    c = Link('Clod', b)
    d = Link('Dian', c)
    e = Link('Eddi', d)
    a.next = b
    b.next = c
    c.next = d
    d.next = e
    for l in forward_links(a):
        print l.data

# result should be:
#    Bob Clod Dian Eddi
```

Doubly Linked List

```
class DoublyLinkedList:
    def __init__(self):
        self.last = None      # tail
        self.first = None     # head
        self.size = 0

    def add_to_back(self, data):
        "Add an item to the tail of the list"

    def add_to_front(self, data):
        "Add an item to the head of the list"

    def remove(self, data):   # Use the two methods below
        "Remove an item from the list"

    def remove_first_item(self):
        "Remove the first item of list"

    def remove_last_item(self):
        "Remove last item of list"

    def items(self):          # List of data items

    def __len__(self):
        return self.size

    def __str__(self):
```

Deque – Double-Ended Queue

- a queue-like data structure that supports insertion and deletion at both the front and the back of the queue
- Methods: `add_fisrt(), add_last(), delete_first(), delete_last(), is_empty(), size(), fisrt(), last()`
- Write an ADT and a basic test (that uses all methods!)
- Can it be implemented using a Python List?
- What about complexity concerns?

Bag Data Structure

- A Bag data structure is exactly as set but duplicates are allowed!
- Write a clear ADT from the test below
- Make sure complexity of operations is super efficient! (try O(1))
- Make sure operations like union, intersection, and difference accept any Python container (list, set, dict, bag, stack, etc.)

```
def bag_test():
    b = Bag([1,2,2])
    b.add(7)
    b.add(2)
    b.add(7)
    b.union([1,5,2,5])           # b = 1,2,2,2,2,5,5,7,7
    b.intersection([2,1,1,2,5,7] # b = 1,1,2,2,5,7
    b.add([1,5,1,5,1,2,2,2])     # how many occurrences of 2 ?
    b.difference([1,2,7,8,16])
    b.remove(2)         # only one instance of 2 is removed
    b.issubset([1,2,3,4,5,6,7])
    print b.items()     # print items with no multiplicity
    b.size()            # count multiplicities
```