ENGR 101 – Chapter 9

Strings, Cells, and Tables

Laura Alford, James Juett, Rick Niciejewski 9/26/2020

Recall: Data Types

- MATLAB supports working with different **types** of data.
- Each type contains a different kind of data and supports a variety of different operations.
- Some examples of types we've seen so far:
  - double – A regular number.
  - uint8 – An integer between 0 and 255, inclusive.
  - char – A character (e.g. 'a')
  - logical – A true or false value, written as 0 or 1.

Scalars, Vectors, and Matrices

- A scalar is a single piece of data of a particular type.
- Arrays (e.g. vectors and matrices) contain several pieces of data, grouped together into a grid.
- Each array must be **homogenous** – it may only contain a single type of data, although it can contain as many pieces of data as we like.
- Arrays must also be "rectangular".
  (You can't have two rows in a matrix of different lengths.)

Recall: The whos function

- The whos function shows us the type of a variable, as well as its dimensions and how much memory it takes to store.

```
>> x = 1;
>> whos x
 Name      Size              Bytes  Class     Attributes
  x         1x1                 8  double

>> y = [1,2; 3,4];
>> whos y
 Name      Size              Bytes  Class     Attributes
  y         2x2                32  double

>> word = ['h', 'e', 'l', 'l', 'o'];
>> whos word
 Name      Size              Bytes  Class    Attributes
  word      1x5                10  char
```

Representing Strings in MATLAB

- A **string** is a sequence of characters (i.e. a "word").
- In MATLAB, a string is simply a vector of chars:

```
>> word1 = ['h','e','l','l','o']
word1 =
hello
>> word2 = 'world'
word2 =
world
>> [word1 ' ' word2]
ans =
hello world
```

Instead of each character individually, we can also use a **string literal**.

Joining strings together is often called **concatenation**.

Storing a List of Strings

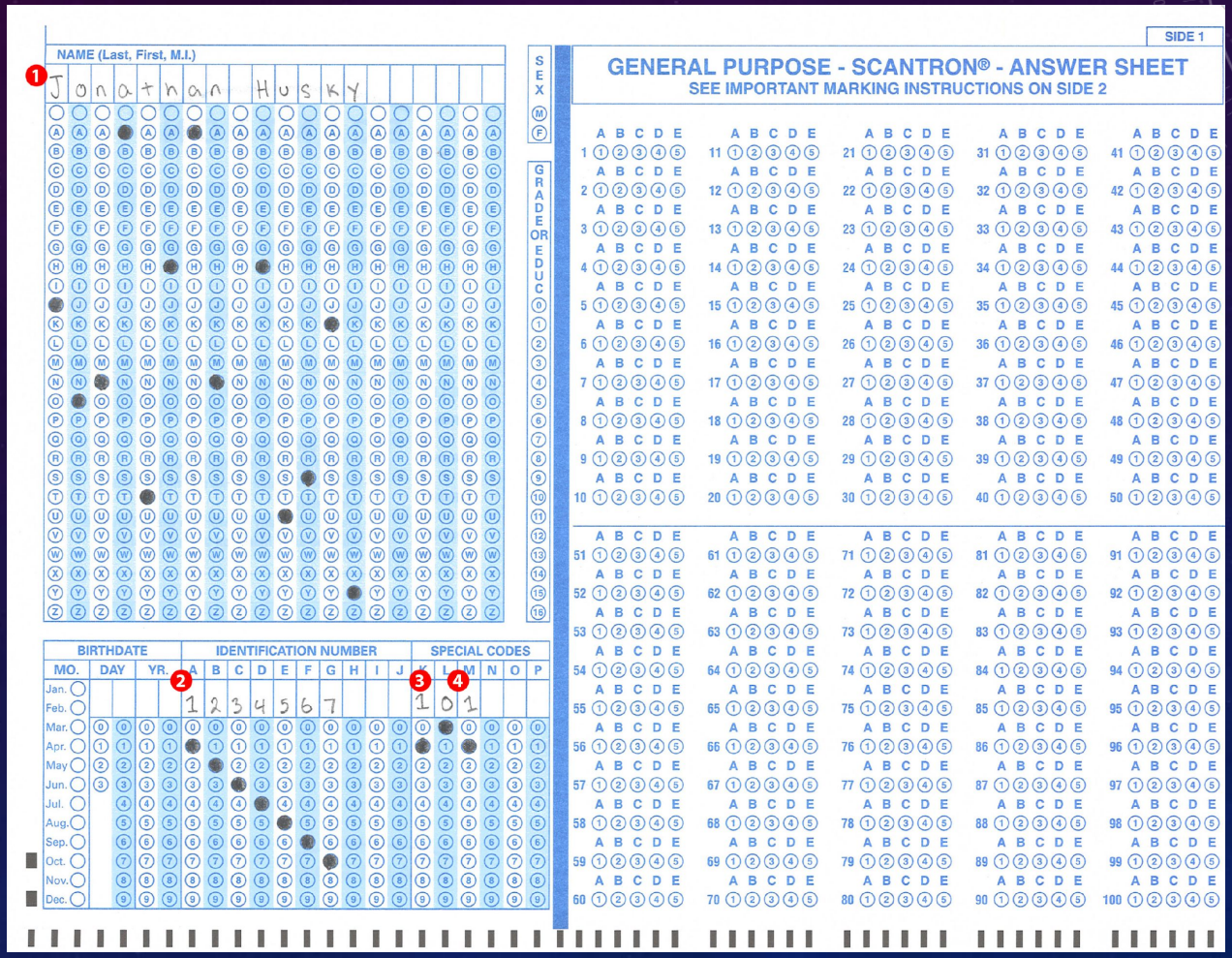
- Let's say we wanted to store several strings, for example, the names of countries in the election day example.
- One option is to use a matrix of characters.
  - Because the matrix must be rectangular, we have to pad with spaces.

'A'	'f'	'g'	'h'	'a'	'n'	'i'	's'	't'	'a'	'n'	' '	' '	' '
'A'	'l'	'b'	'a'	'n'	'i'	'a'	' '	' '	' '	' '	' '	' '	' '
'A'	'l'	'g'	'e'	'r'	'i'	'a'	' '	' '	' '	' '	' '	' '	' '
'A'	'm'	'e'	'r'	'i'	'c'	'a'	'n'	' '	'S'	'a'	'm'	'o'	'a'
'A'	'n'	'd'	'o'	'r'	'r'	'a'	' '	' '	' '	' '	' '	' '	' '

5x14 char

  - One problem: If one string is a lot longer than the others, we have a lot of wasted space!
  - Another problem: the extra spaces get in the way!

You've seen things like this before



Cell Arrays

- In MATLAB, a **cell** array allows us to create a **heterogeneous** collection of elements.
- All elements in a cell array are of type "cell", but a cell may subsequently refer to any other type of data.
- Use the curly brackets { and } to create a cell array.

```
>> test = {1, 'hello', [1,2,3]}

test =

    [1]    'hello'    [1x3 double]
```

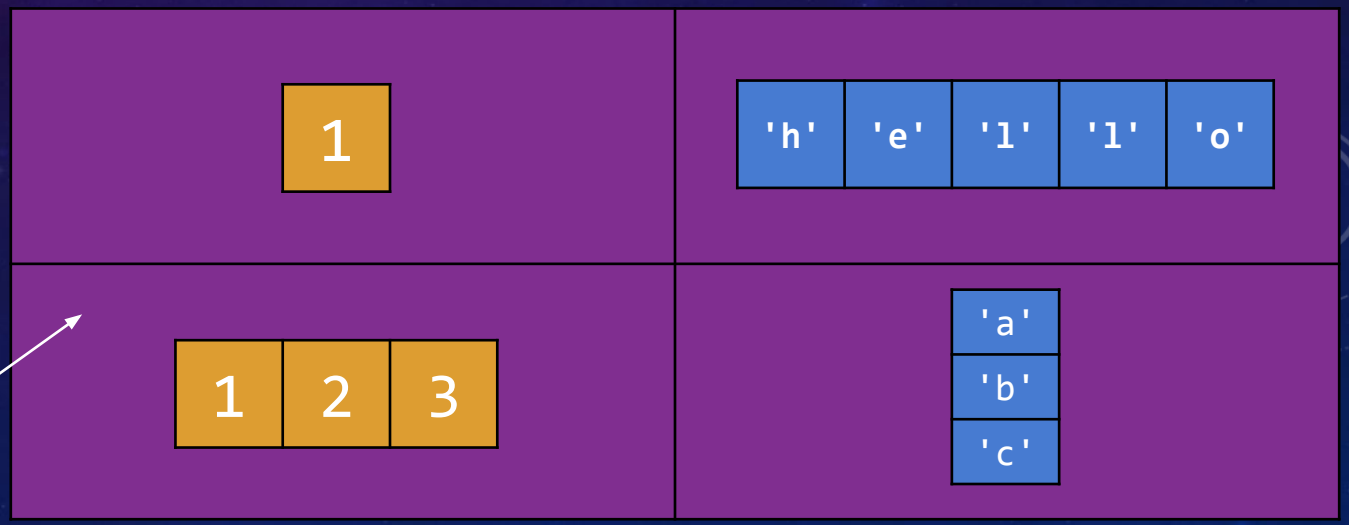
Cell Arrays

- The syntax for creating cell arrays is similar to normal arrays.

```
>> test = {1, 'hello'; [1,2,3], ['a';'b';'c']}
  2×2 cell array
    {[       1]}    {'hello' }
    {1×3 double}    {3×1 char}
```

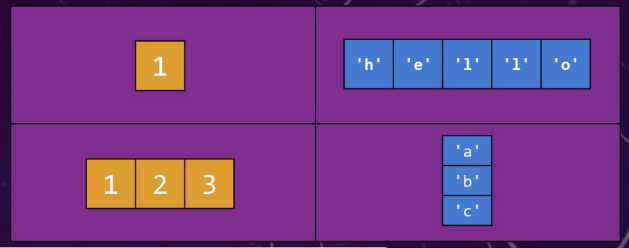
For convenience, MATLAB will show the contents of cells containing strings (i.e. vectors of char).

MATLAB shows each cell in the cell array, as well as the type of data inside the cell.



The cell array is nice and "rectangular", but the contents of each cell can be whatever we want.

Cell Indexing with ( )




```
test = {1, 'hello'; [1,2,3], ['a';'b';'c']};
```

- Indexing with ( ) works the same as usual:

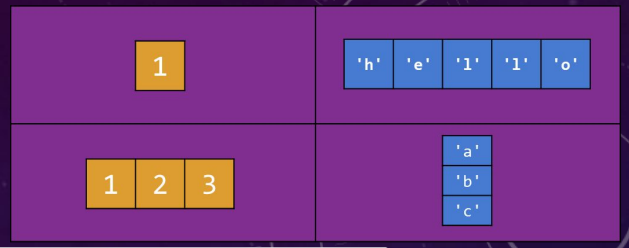
```
>> test(2,1)
ans =
  1×1 cell array
    {1×3 double}
```

The cell at row 2 col 1.


```
>> test(2,:)
ans =
 1×2 cell array
  {1×3 double}  {3×1 char}
```

The second row of cells.

Content Indexing with { }




```
test = {1, 'hello'; [1,2,3], ['a';'b';'c']};
```

- Indexing with { } automatically *unpacks* the cell to give you its contents:

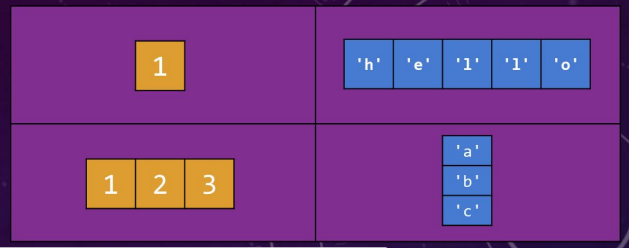
```
>> test{2,1}

ans =
     1     2     3
```

The vector [1, 2, 3]

```
>> test{1,2}

ans =
    'hello'
```

The string 'hello' (a 5x1 char vector)

Indexing with ( ) vs. { }




```
test = {1, 'hello'; [1,2,3], ['a';'b';'c']};
```

- If you want to work with the data, usually you'll need to unpack it from the cell with { }.

```
>> test(2,1) + 1
```

+ 1

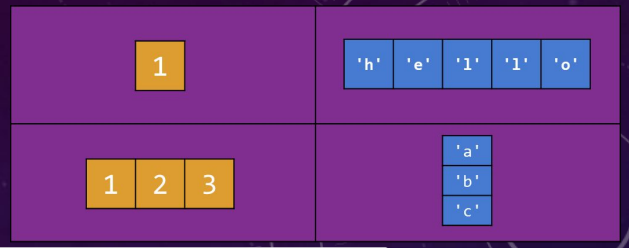
ERROR. Can't "add 1 to a cell".

```
>> test{1,2} + 1
```

1 2 3 + 1

ans = 2 3 4

Indexing with ( ) vs. { }




```
test = {1, 'hello'; [1,2,3], ['a';'b';'c']};
```

- If you want to select a subarray of cells, use ( ).
  You don't need to unpack the data for this.

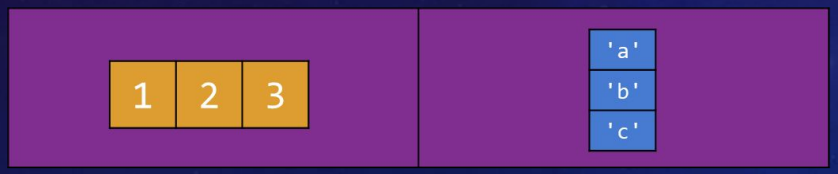
```
X = test(2,:)
```

YES

```
X = test{2,:}
```

NO

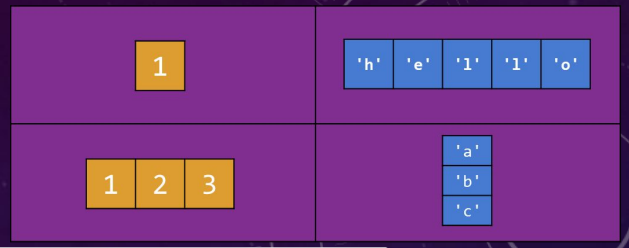
X is now



X is now

GENERALLY NOT VERY USEFUL

Unpacking to Individual Variables




```
test = {1, 'hello'; [1,2,3], ['a';'b';'c']};
```

- If you unpack a selection to individual elements, use { } and specify a target variable for each.

```
X = test{2,:}
```

X is now

2	3	4

Just gives you the first one.

```
[X, Y] = test{2,:}
```

X is now

2	3	4

Y is now

'a'
'b'
'c'

A variable for each.

Unpacking Cells to a Matrix


```
test = {1, 'hello'; 5, ['a';'b';'c']};
```

- If you want to unpack *many* elements, use ( ) to select a subarray of cells, and then the `cell2mat` function.
- But – make sure the data in the cells will play nicely!

```
X = cell2mat(test(1,:))
```

X is now

1	5

```
Y = cell2mat(test(2,:))
```

ERROR. Can't merge a 1x3 double vector and a 3x1 char vector into something sensible.

Converting Between Regular and Cell Arrays

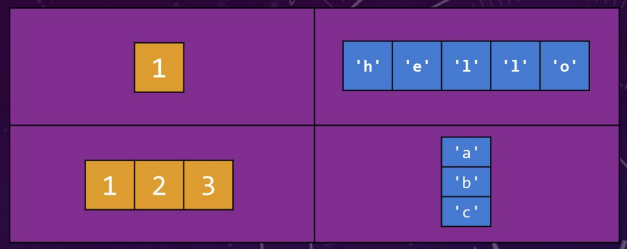
- To create a cell array from a regular array of numbers:

```
C = num2cell(A)
```

- To create a regular array from a cell array containing numbers:

```
A = cell2mat(C)
```

Recap: Indexing Into Cell Arrays




```
test = {1, 'hello'; [1,2,3], ['a';'b';'c']};
```

- There are two different ways to indexing into a cell array:

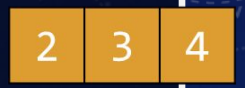
- **Cell Indexing** – uses ( )
  Selects the cells themselves.
  Result is a smaller cell array.

```
>> test(2,1)
 1×1 cell array
   {1×3 double}
```

A cell (which contains the vector)

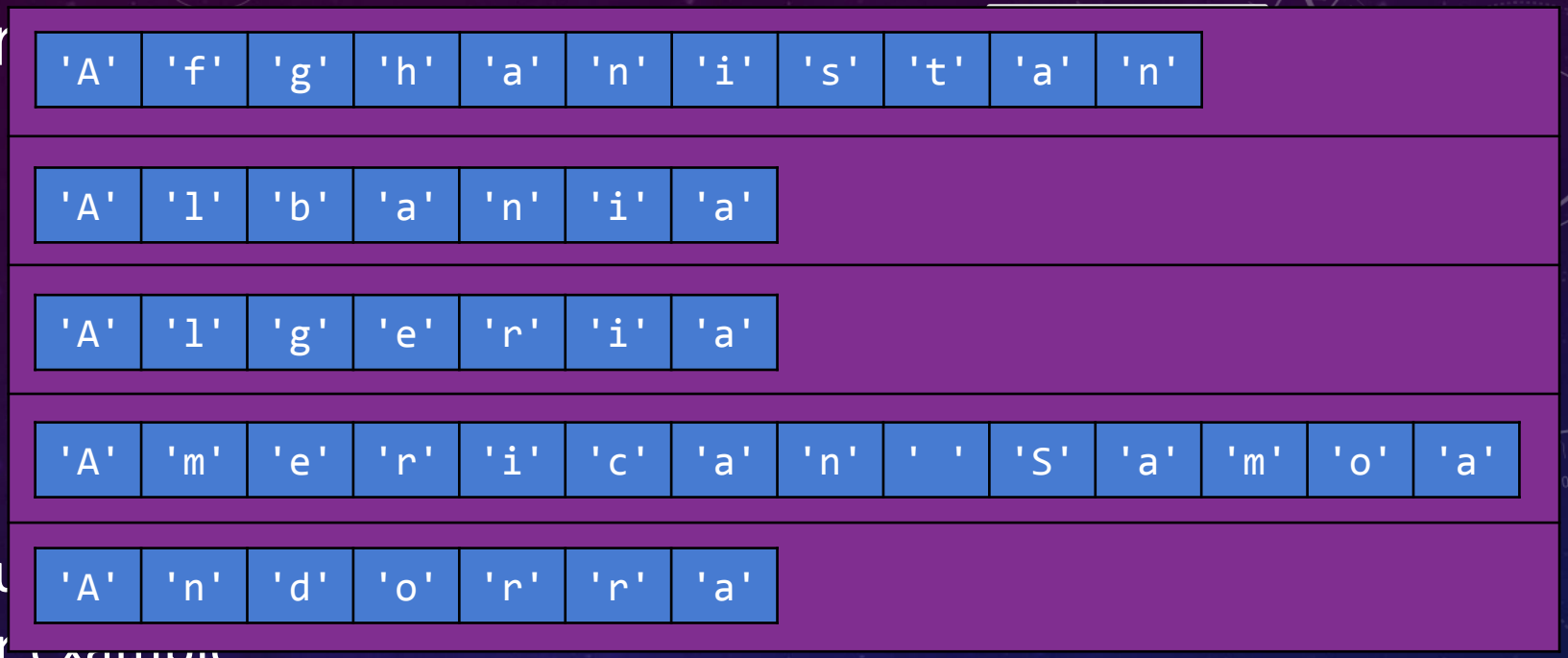
- **Content Indexing** – uses { }
  Unpacks the contents of the cells (i.e. the real data).
  Generally not useful for multi-element selections.

```
>> test{2,1}
ans =
     1     2     3
```



A 1x3 double vector

Stor



- You
  For example:

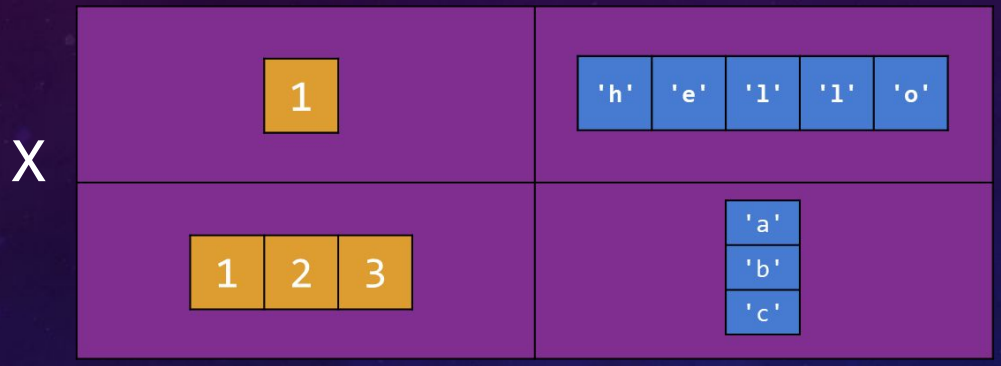
```
>> [states(3,:), ' hello']
ans =
Algeria        hello

>> [deblank(states(3,:)), ' hello']
ans =
Algeria hello
```

The deblank function removes trailing spaces.

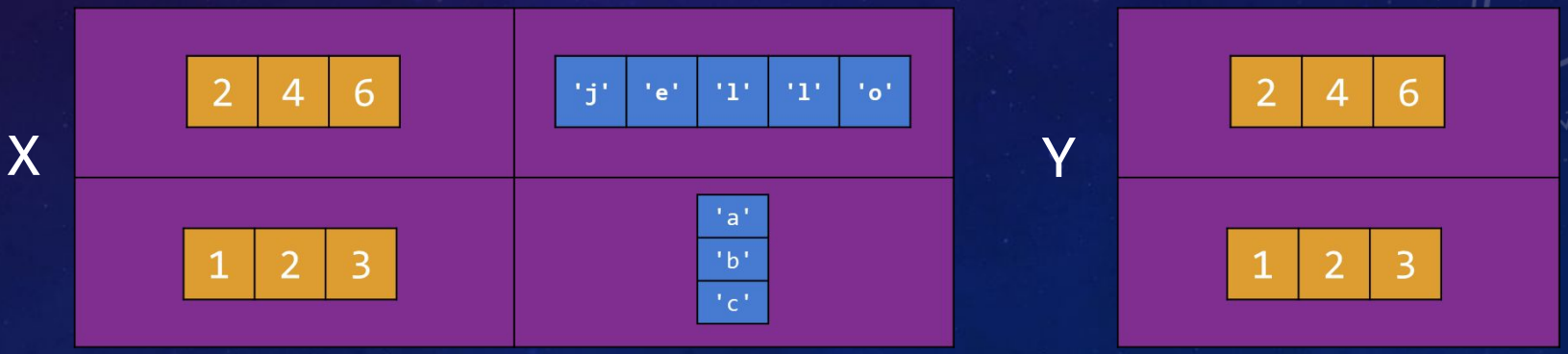
Exercise: Practice with Cell Arrays

- Start with this variable X, which stores a cell array:

```
X = {1, 'hello'; [1,2,3], ['a';'b';'c']}
```



- Turn it into this:

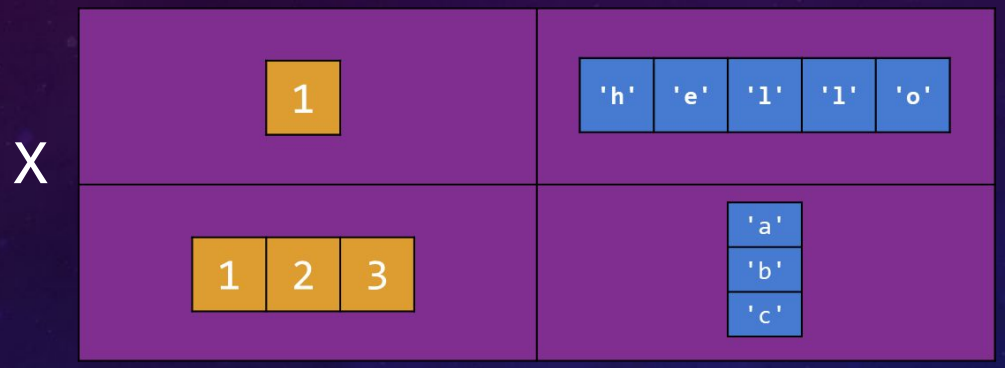


Hint: you can use the variable window to check your progress.

Solution: Practice with Cell Arrays

- Start with this variable X, which stores a cell array:

```
X = {1, 'hello'; [1,2,3], ['a';'b';'c']}
```

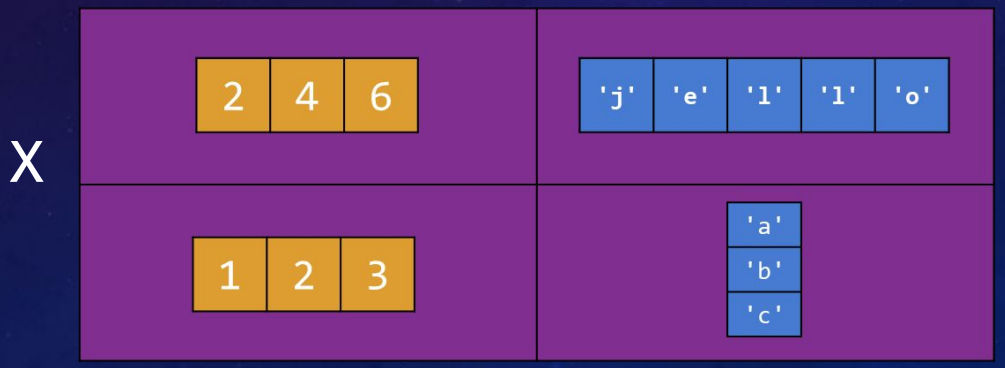


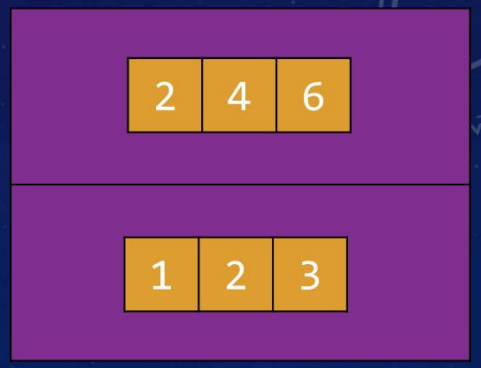

```
X{1,2} = 'jello';
        OR
X(1,2) = {'jello'};
        OR
X{1,2}(1) = 'j';
```

```
X{1,1} = 2 .* X{2,2};
```

```
Y = X(:,1);
```

- Turn it into this:

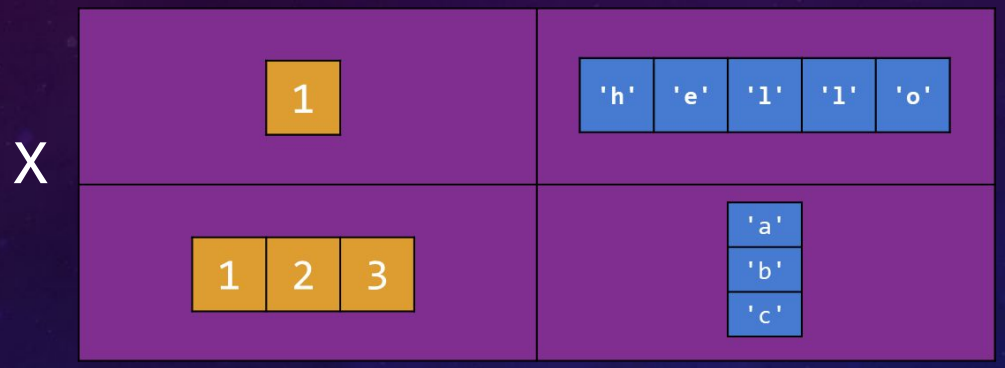




Solution: Practice with Cell Arrays

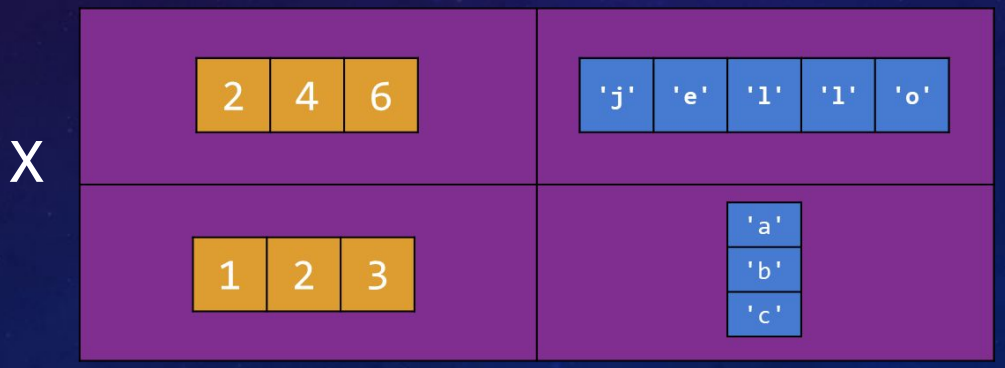
- Start with this variable X, which stores a cell array:

```
X = {1, 'hello'; [1,2,3], ['a';'b';'c']}
```



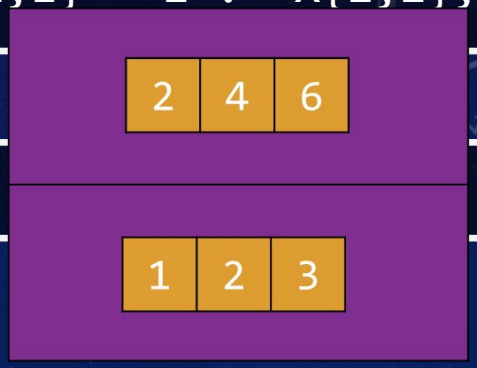

```
X{1,2} = 'jello';
        OR
X(1,2) = {'jello'};
        OR
X{1,2}(1) = 'j';
```

- Turn it into this:




```
X{1,1} = 2 .* X{2,2};
```

Y



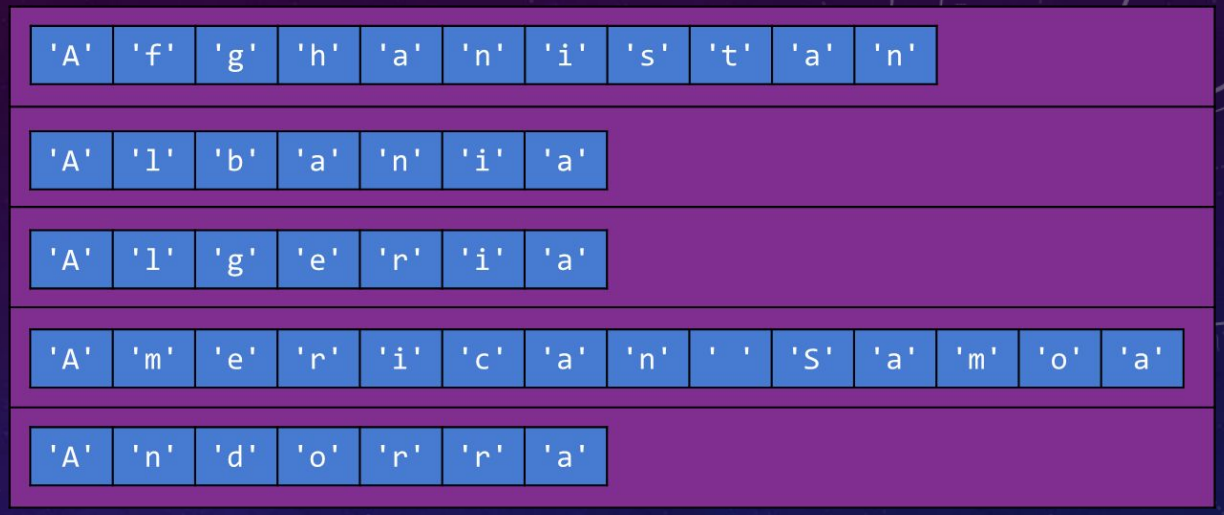
When to Use Cell Arrays?

- Cell arrays allow for a heterogeneous vector/matrix of elements of different types.
  - This is not a common need.
  - Always prefer regular arrays if they will do the job.

- Example 1: Working with strings (as char vectors of potentially different lengths).

Storing a List of Strings

```
states - 5x1 cell
(column vector)
```



- More memory-efficient and no annoying extra spaces!

```
>> states{3}
ans =
Algeria
```

```
>> states(1:3)
ans =
  3×1 cell array
    {'Afghanistan'}
    {'Albania'    }
    {'Algeria'    }
```

String Concatenation

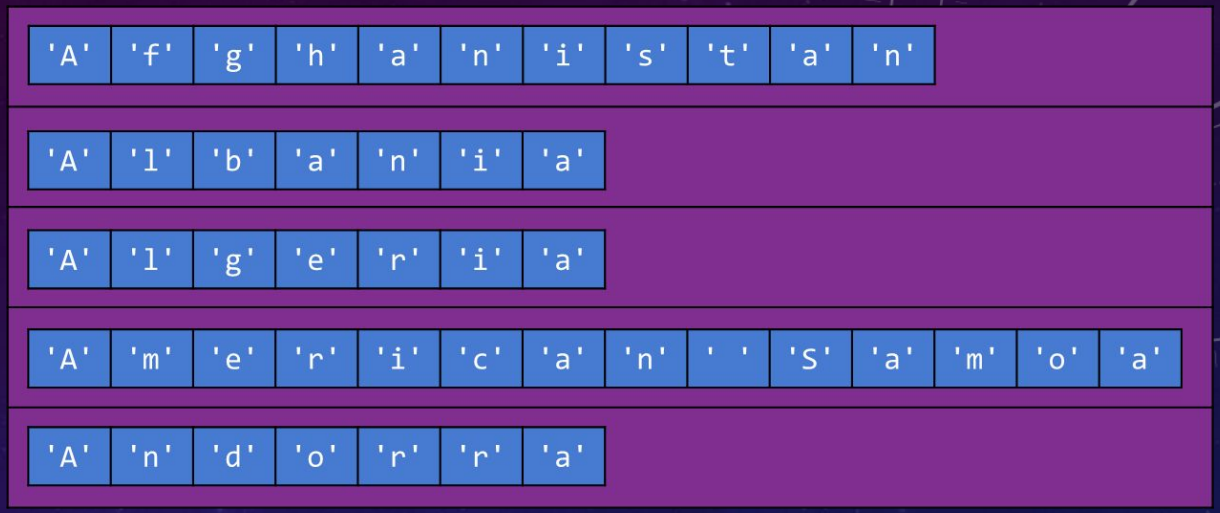
```
str1 = 'hello';
str1 = 'world';
```

- Use the `strcat` function (recommended method)

```
>> strcat(str1, str2)
ans =
helloworld

>> strcat(str1, ' ', str2)
ans =
Hello world
```

Vectorized String Concatenation

`states - 5x1 cell (column vector)`



- MATLAB knows it's a good idea to store strings in cells. Built-in string functions are vectorized and work with cell arrays of strings!

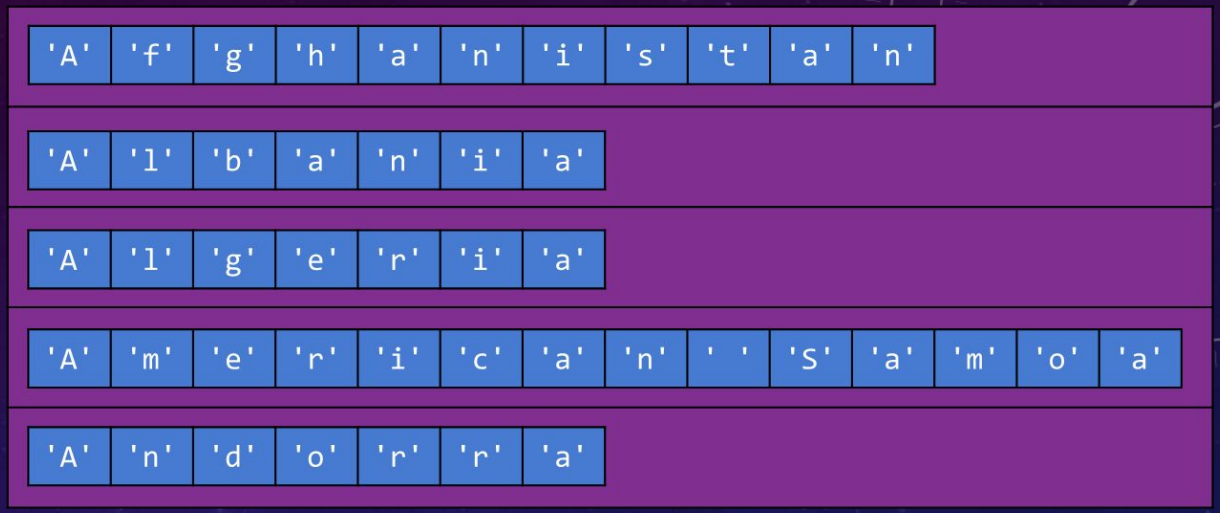
```
>> strcat({'Hi '}, states(1:3))
ans =
  3×1 cell array
    {'Hi Afghanistan'}
    {'Hi Albania'    }
    {'Hi Algeria'    }
```

You don't have to "unpack" the strings from the cells before using them!

*Note the { } around `{'hi '}` are technically necessary to preserve the space. `strcat()` by default trims whitespace unless strings are in a cell.

Vectorized String Comparison

states - 5x1 cell
(column vector)



- The == operator won't work correctly by default for strings, because it tries to do an element-by-element operation. Use `strcmp()` instead.

```
>> strcmp('Albania', states)
ans =
  5×1 logical array
   0
   1
   0
   ...
```

Double Quote Strings

- There are actually two different kinds of strings in MATLAB:

Single quotes

```
>> x = 'hello';
>> whos x;
 Name  Size  Bytes  Class
 x     1x5      10  char
```

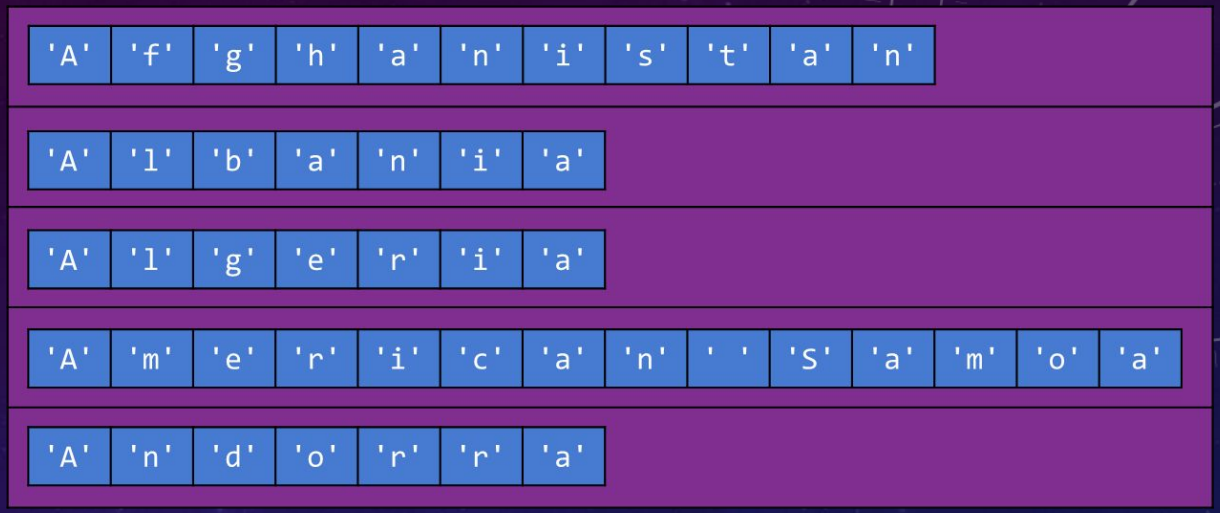
Double quotes

```
>> y = "hello";
>> whos y;
 Name  Size  Bytes  Class
 y     1x1    150  string
```

- Single quote strings are often the kind you end up with initially after e.g. reading in data from a file.
- Double quote strings have some nice convenience features.
  - Use them when you can!

Double Quote String Features

states - 5x1 cell
(column vector)



- Many operators that do not work with single quote strings will work if at least one of the operands is a double quote string:

```
>> states == "Albania"
ans =
  5×1 logical array
   0
   1
   0
   ...
```

```
>> states < "B"
ans =
  5×1 logical array
   1
   1
   1
   ...
```

```
>> "Hi " + states
ans =
  5×1 string array
   "Hi Afghanistan"
   "Hi Albania"
   "Hi Algeria"
   ...
```

Break Time

We'll start again in 5 minutes.

When to Use Cell Arrays?

- Cell arrays allow for a heterogeneous vector/matrix of elements of different types.
  - This is not a common need.
  - Always prefer regular arrays if they will do the job.

- Example 1: Working with strings (as char vectors of potentially different lengths).
- Example 2: Reading from a data file that contains both text and numeric data (e.g. an Excel spreadsheet).

xlsread

City	Population	Latitude	Longitude
Shanghai	24,256,800	31.20	121.50
Karachi	23,500,000	24.87	67.02
Beijing	21,516,000	39.90	116.40
Delhi	16,349,831	28.62	77.22
Lagos	16,060,303	6.45	3.40

- The `xlsread` function reads data from Microsoft Excel files, which generally have the `.xls` or `.xlsx` extension.
- Several optional parameters customize its behavior
  - See the documentation for full details.
- `xlsread` uses a compound return for numeric and text data.

```
[num, txt, raw] = xlsread('cities.xlsx');
```

A regular array of only the numeric data.

A cell array containing only the text data.

A cell array containing all the data. (Cells allow us to combine numeric and text data.)

Tables in MATLAB

- Topics covered:
  - Reading in a table
  - Fundamentals – each column is either a vector or cell vector
  - Indexing vs. unpacking
  - Height, width, size
  - Adding a row
  - Adding a column
  - Sorting
  - Indexing