PSYCH 20A (PROF. FRANE) – LECTURE 6
Cell Arrays, Structures, Loading/Saving, and Tables

Cell arrays

Unlike the arrays we've been discussing so far, the elements of a cell array aren't values. The elements of a cell array are containers called *cells*. Each cell can contain any object we want—a string, a number, an array, whatever. We can stuff an object of any class and size into a cell.

Making an array of cells is similar to making an array of values, but we use curly brackets `{}` instead of square brackets `[]`. For example, to make a 3 × 1 cell array called `myFaveFruits`:

```
myFaveFruits = {'apple' ; 'pear' ; 'avocado'}
```

Indexing a cell array

Note that `myFaveFruits` as defined above is a column-vector with 3 cells in it. The first element in `myFaveFruits` is not actually the character string `apple`, but rather a cell—a container—that has the character string `apple` inside it.

If we want to access the character vector `apple` itself, we need to "extract" it from the cell. We do that by using curly brackets instead of parentheses when indexing, like this: `myFaveFruits{1}`

To be clear, `myFaveFruits(1)` is a single cell (and thus has a size of 1 × 1), whereas `myFaveFruits{1}` is a 1 × 5 character array sitting inside that cell.

Extracting multiple objects from a cell array into an array

We discussed extracting a single object from a cell using curly brackets `{}`. We can extract some or all of the contents of a cell array at once into an array using the `cell2mat` function, as long as all the extracted objects are the same type (e.g., all numeric or all characters) and the number of rows and columns are consistent.

First let's define a 1 × 6 cell array in which each cell contains either a character array, a number, or a vector:

```
myCellArray = {'red', 'yellow', 'purple', 34, 21, 15:20}
```

We wouldn't be able to extract the contents of all the cells into a single array, because some of the objects are characters and some are numbers. But we could convert the character-array portion and the numeric portion separately:

`cell2mat(myCellArray(4:6))` would output the following 8-element row vector:
34 21 15 16 17 18 19 20

`cell2mat(myCellArray(1:3))` would output the following 15-character row vector:
redyellowpurple

Many "string" functions can also be applied to a cell array of character vectors, e.g.:

`string(myFaveFruits)`	Outputs the string column-vector ["apple" ; "pear" ; "avocado"]'
`strjoin(myFaveFruits, '--')`	Outputs the character row-vector 'apple--pear--avocado'
`strlength(myFaveFruits)`	Outputs the numeric column vector [5 4 7]'
`upper(myFaveFruits)`	Outputs the cell column-vector {'APPLE' ; 'PEAR' ; 'AVOCADO'}
`str2double({'10' '15'})`	Outputs the numeric row-vector [10 15]

Making an array of empty cells

Sometimes we want to *initialize* a cell array as a bunch of empty cells that we will fill in later. We can do that by inputting the desired dimensions of the array to the `cell` function. For example, to make a 5 × 1 array of empty cells called `x` you would type the following: `x = cell(5, 1)`

Structures ("structs")

It is often useful to package different types of data together into a single object, e.g., so we can store all the data from the same subject in a single object. One way to do that is with a cell array. Recall that a cell array is an array of cells (containers), each of which can hold an object of any type and size. But a drawback of cell arrays is that the cells don't have labels that describe their contents; they are only indexed by their position in the array. Thus, we would have to remember that, for example, cell 1 contained the subject's sex, cell 2 contained the subject's age, cell 3 contained a vector of test scores for the subject, etc.

Another construction that can hold multiple types and sizes of objects inside it is a *structure* (a "struct"). A structure is like a big bag that can hold all the objects we want. Instead of putting those objects in containers and arranging them in a specific order, we label the objects using names ("sex," "age," "testScores," etc.) that we call *fields*. Unlike in an array, there's no particular ordering/arrangement to the fields in a structure—we don't need the objects to have numeric indexes, because we can just refer to the objects by name.

Defining a structure from scratch

Let's say we want to make a structure called `myData`, which contains three fields: sex, age, and a vector of test scores. We can define each field of the structure separately by using periods:

```
myDataStruct.sex        = 'M' ;
myDataStruct.age        = 23 ;
myDataStruct.testScores = [88 75 78 96] ;
```

Or we could define that structure by using the `struct` function, which we won't cover in detail here.

When a field in the structure is an array, we can index it just as we could with any other array. For example, to get the 3rd value in `myDataStruct.testScores` we can type `myDataStruct.testScores(3)`

Defining a structure from a cell array

If we've already defined a cell array, we can convert it to a structure by entering that cell array, followed by a corresponding cell array of field names, to the `cell2struct` function. For example, let's say we've defined the following 3 × 1 cell array (by default, `cell2struct` expects the cells to be in a column):

```
myDataCellArray = {'M' ; 23 ; [88 75 78 96]} ;
```

Then we can convert it to a structure like this:

```
myDataStruct = cell2struct(myDataCellArray, {'sex' 'age' 'testScores'}) ;
```

Structure arrays

What if instead of having several types of data from just one subject, we have several types of data from each of multiple subjects? In that case, we might want to organize all that information into a *structure array*. A structure array can be indexed simply using parentheses. The following code defines a structure array called `testData`, which contains two structures (one for each of two subjects):

```
testData(1).subjectNumber = 1 ;
testData(1).sex           = 'M' ;
testData(1).age           = 23 ;
testData(1).testScores    = [88 75 78 96] ;

testData(2).subjectNumber = 2 ;
testData(2).sex           = 'F' ;
testData(2).age           = 21 ;
testData(2).testScores    = [77 78 89 94] ;
```

The fields (`subjectNumber`, `sex`, `age`, `testScores`) must be the same for each structure in the array.
To get the age of the 1st subject, we can type `testData(1).age`
To get the 4th test score for the 2nd subject, we can type `testData(2).testScores(4)`

Saving and loading objects

Imagine that we have a matrix called `dataMat` and a cell array called `myCellArray` that we want to save so we can use them later. The following command will save both of those objects in the present working directory under the filename `myMatrixFile.mat`

```
save('myMatrixFile', 'dataMat', 'myCellArray')
```

Note that the first input to the save function is the filename (in quotes; a `.mat` extension will be added automatically if you leave it off), and the rest of the inputs are the names of the objects we're saving (in quotes). Later, if we want to recover `dataMat` and `myCellArray` we can do that like this:

```
load 'myMatrixFile'
```

If we want to save the file somewhere other than the current working directory, give a full path (don't forget the quotes), e.g.:

```
save('/Users/UserName/Downloads/myMatrixFile', 'dataMat', 'myCellArray')
```

Remember, the exact path will depend on your specific system (e.g., on a PC it may start with `c:`).
You can also load from a specified location, as follows:

```
load '/Users/UserName/Downloads/myMatrixFile'
```

To save *all* the variables defined in the workspace under a single filename, use the `save` function with a filename and no second input:

```
save('/Users/UserName/Downloads/myWorkspaceFile')
```

Tables

Importing a table from a csv file

Sometimes you want to import a table from data that's stored in a csv file that was created by another program, such as Excel. There are many ways to do this. The simplest way is to use the `readtable` function. The following code creates a table called `myTable` from a file called `someDataFile.csv` :

```
myTable = readtable('someDataFile.csv') ;
```

Or if the file is stored somewhere other than the current working directory:

```
myTable = readtable('/Users/Username/Documents/someDataFile.csv') ;
```

The elements in a given column must have the same class. By default, any column that has non-numeric data will be formatted as a cell array when imported. And the class of any column with numeric data is set automatically based on the contents (e.g., integers or double-precision values). We can override those defaults by adding `'Format'` as the 2nd input to the `readtable` function, and then adding a character array (or string) as the 3rd input giving the formatting codes for the respective columns. For example, if the first two columns contain continuous numeric measurements, and the third column contains character-codes of uniform length (such as 'M' and 'F' values indicating subject's sex):

```
myTable = readtable('someDataFile.csv', 'Format', '%f%f%c') ;
```

`%f` indicates "fixed point" decimal numbers (as opposed to, e.g., integers), `%d` allows Matlab to choose the numeric format automatically, and `%c` indicates a character array (see the documentation for other formatting codes, e.g., for dates). If the table has row names, add `'ReadRowNames', true` as inputs to `readtable`.

Indexing tables using 2-dimensional indexing

Imagine that the following table is called `subjectData`:

Age	Weight	Sex
19	197	'Male'
18	145	'Female'
22	180	'Female'
18	177	'Male'

We can index the table using 2-dimensional indexing, just like for a matrix except we use curly brackets `{}` instead of parentheses `()`. For the column index we can use either a number or the name of the column. For example, to output the weight of the 3rd subject, we could type either of the following lines:

```
subjectData{3, 2}
subjectData{3, 'Weight'}
```

And to get the mean of all the ages, we could type either of the following lines:

```
mean( subjectData{:, 1} )
mean( subjectData{:, 'Age'} )
```

To get a vector of all the ages that are under 20, we can use logical indexing:

```
subjectData{ subjectData{:, 'Age'} < 20, 'Age' }
```

To get a vector of all the weights for subjects whose ages are under 20:

```
subjectData{ subjectData{:, 'Age'} < 20, 'Weight' }
```

To delete an entire column from a table, set the column equal to an empty vector (here we use parentheses, not curly brackets, because we are deleting the column itself, not the individual values in the column):

```
subjectData(:, 'Age') = [] ;
```

Indexing tables like structs

Tables can also be indexed like a structure: Type the name of the table, followed by a period, followed by the column name, followed by the row indexes in parentheses (or put the row indexes in curly brackets if the column is a cell array). That makes the syntax simpler. For instance, using the same table as above, to get the 3rd value in the Age column:

```
subjectData.Age(3)
```

To extract the sex of the 3rd subject, we would use curly brackets `{}` because the Sex column is a cell array:

```
subjectData.Sex{3}
```

To get a vector of all the weights for subjects whose ages are under 20:

```
subjectData.Weight(subjectData.Age < 20)
```

To get a vector of all the weights for the male subjects:

```
subjectData.Weight( strcmp(subjectData.Sex, 'Male') )
```

Converting a table to an array

To convert an entire table to a matrix, we can use the `table2array` function. For the above table, we wouldn't be able to convert all the columns because they aren't all the same class. But we could convert just the first two columns:

```
subjectDataAsMatrix = table2array(subjectData(:, 1:2)) ;
```

Creating a table from scratch

Let's say we want to make the above table ourselves. First we make the matrix, like this for example (note that we use the apostrophe to transpose the row-vectors into column-vectors; alternatively, we could use semicolons between the values to make it a column vector in the first place):

```
age    = [ 19  18  22  18]' ;
weight = [197 145 180 177]' ;

ageWeightMatrix = [age weight] ;
```

Then we convert the matrix to a table and add column names, using the `array2table` function (the column names can be in either a string array or a cell array of character vectors):

```
ageWeightTable = array2table(ageWeightMatrix, 'VariableNames', ["Age" "Weight"]);
```

In some cases, we might want row names as well (like the column names, the row names can be in either a string array or a cell array of character vectors:

```
ageWeightTable = array2table(ageWeightMatrix, ...
'VariableNames', ["Age" "Weight"], 'RowNames', ["A" "B" "C" "D"]) ;
```

Note that the lists of row and column names are input as string arrays (cell arrays will also work).
The "`...`" in the first line just means that the command continues on the next line; this is handy when you have a long piece of code that otherwise wouldn't fit on the screen (but you can't use "`...`" that way inside a string or character array, because Matlab would think it was part of the text).

The `cell2table` function is similar to `array2table`, but takes a cell array as input (which is handy when you can't put all the columns in a matrix because they aren't all the same class of values).

Adding columns to an existing table

We can add columns by using 2-dimensional indexing:

```
ageWeightTable{:, "Height"} = [62 ; 74 ; 70 ; 73] ;
```

Or by using a period (like for a field in a struct):

```
ageWeightTable.WeightOverHeight = ageWeightTable.Weight ./ ...
                                  ageWeightTable.Height ;
```

Saving/exporting a table

You can save a table or structure as a Matlab object (`.mat` file) by using the `save` function, the same way you would save a matrix or a cell array or a structure or any other Matlab object. For example:

```
save('c:/Users/UserName/Documents/ageWeightTableFile', 'ageWeightTable')
```

Of course, the path on your system will be different, and if you're saving to the present working directory, you can just enter the filename instead of the full path.

We can also export the table as a csv file, which is handy for sharing with collaborators who don't use Matlab. We can do that using the `writetable` function:

```
writetable(ageWeightTable, 'ageWeightFile.csv')
```

Note that in the `save` function, the filename (in quotes) is the first argument and the name of the object (in quotes) is the second argument, whereas in the `writetable` function, the object itself (without quotes) is the first argument and the filename (in quotes) is the second argument.

If you have row names in your table, they won't be saved in the csv file by default. To save the row names (in a column called "Row"), turn on the `'WriteRowNames'` setting when you use `writetable`:

```
writetable(ageWeightTable, 'ageWeightFile.csv', 'WriteRowNames', true)
```