

# Customizable tables and reproducible reports using Stata

Chris Cheng

StataCorp LLC

Chicago, IL  
ASA Teaching Workshop  
August 10, 2025

# Outline

- 1 Intro
- 2 The table command
- 3 The dtable command
- 4 The etable command
- 5 The collect system
- 6 Reproducible reports

# Intro

- There are four ways to make customizable tables in Stata: `table`, `etable`, `dtable`, and the `collect` suite of commands. These commands can also work together, as we will see.

# Intro

- The `table` command allows you to create various types of tables.
- The `etable` command allows you to quickly create a table of estimation results following an estimation command.
- The `dtable` command allows you to quickly create a Table 1 of descriptive statistics.
- The `collect` suite can do all the above and more. The `collect` suite uses a series of commands to create tables and export them step by step.

# Intro

- After we master creating tables, we can write reproducible reports in Stata for Word documents, Excel spreadsheets, HTML webpages, and more.

## Creating tables with the table command

# The table command

- We can create tables using any combination of three types of information: cross-tabulations of variables, summary statistics, and results returned by commands. We will see examples of each of these individually and examples where we combine them. Later, we will discuss how to format and style for your tables.

# The table syntax

- The basic syntax of the `table` command is

```
table (rowspec) (colspec) (tablespec) [, options]
```

- In other words, we need to specify what identifies the rows, what identifies the columns, and what identifies different tables (if desired).



# Load dataset

- Let's load a new dataset that will give us many different types of variables to create tables from.

```
. sysuse nlsw88, clear
```

```
. describe
```

# Cross-tabulations

- Tabulations allow you to examine the distribution of your data across the levels of one or more categorical variables.

# One-way tabulations

- To obtain a one-way tabulation that reports the number of observations for each level of a categorical variable, we need to specify only the name of the variable following `table`.

```
. table union
```

- If we don't want this row in our table, we can add the `nototals` option.

```
. table union, nototals
```

- If we want to report the number of missing responses, we can add the `missing` option.

```
. table union, nototals missing
```

# One-way tabulations

- If you have a variable with many levels, you can use brackets to show only the levels you are interested in.

```
. table industry
```

```
. table industry[2 4 6]
```

- Notice that we use the numeric values, not the value labels, in brackets. To see a list of value labels for a variable, you can use `labelbook` or `label list`.

```
. label list indl1
```

# Two-way tabulations

- We can create two-way tabulations by simply adding another variable to our table command.

```
. table industry union
```

```
. table industry[2 4 6] union
```

- Notice that there are no mining workers in unions. If we would like this cell to have a zero rather than be blank, we can add the `zerocounts` option.

```
. table industry[2 4 6] union, zerocounts
```

# Two-way tabulations

- Returning to our table on union membership, let's add `smsa` to get the counts of union workers by metropolitan area.

```
. table union smsa  
  
. table union smsa, total(smsa)  
  
. table (union) (smsa)  
  
. table union () smsa, nototals
```

# Two-way tabulations

- We also could have specified both `union` and `smsa` as the rows (or the columns). When we have multiple specifications for one dimension, order matters.

```
. table (union smsa)
```

# Multiway tabulations

- We can make tabulations with any number of variables.

```
. table union smsa collgrad
```

```
. table union (smsa) (collgrad), nototals
```

- Here we've nested south by smsa in the rows and collgrad by union in the columns.

```
. table smsa south (union collgrad), nototals
```



# Tables with summary statistics

- In addition to reporting tabulations of frequency counts, we can use `table` to report summary statistics by specifying the `statistic()` option. There are currently around 38 summary statistics available, and `statistic()` may be repeated to request multiple statistics.
- Type `help table` to see the full list.

# Frequency and ratio statistics

- As we have seen, frequency is the default summary statistic in table.

```
. table smsa union, statistic(frequency)
```

```
. table smsa union, statistic(percent)
```

- We can add `statistic(frequency)` to get frequencies and percents.

```
. table smsa union, statistic(frequency) statistic(percent)
```

# Summary statistics

- The summary statistic options require a *varlist*. For example,

```
. table south, statistic(mean hours wage)
```

- If we still wanted to see frequency counts for south, we can add `statistic(frequency)`.

```
. table south, statistic(frequency) statistic(mean hours wage)
```

- We can add as many `statistic()` options as we want.

```
. table south, statistic(frequency) statistic(percent) ///  
statistic(mean hours wage) statistic(sd hours wage)
```

## Tables with command results

- The `table` command can be used to create tables with results of hypothesis tests and models.

# Hypothesis tests

- By default, `table` will choose some results from the specified command to put in a table. In a regression model, the default returned results are the regression coefficients.

```
. table, command(regress hours tenure)
```

- Some commands will display a lot of results.

```
. ttest wage, by(union)
```

```
. return list
```

```
. table, command(ttest wage, by(union))
```

# Hypothesis tests

- Let's add some results to our table.

```
. table, command(r(mu_1) r(mu_2) r(t) r(p): ttest wage, by(union))
```

- We've created a table with each group's means,  $t$  test, and  $p$ -value. We can give names to each of our results.

```
. table, command(Nonunion=r(mu_1) Union=r(mu_2) r(t) r(p): ///  
ttest wage, by(union))
```

- We can combine multiple commands into one table. Here we're adding the  $t$  test on hours worked.

```
. table, command(Nonunion=r(mu_1) Union=r(mu_2) r(t) r(p): ///  
ttest wage, by(union)) ///  
command(Nonunion=r(mu_1) Union=r(mu_2) r(t) r(p): ///  
ttest hours, by(union))
```

# Hypothesis tests

- At this point, we may want to make some layout changes.

```
. table result command, command(Nonunion=r(mu_1) ///  
Union=r(mu_2) r(t) r(p): ttest wage, by(union)) ///  
command(Nonunion=r(mu_1) Union=r(mu_2) r(t) r(p): ///  
ttest hours, by(union))
```

# Regression

- When we specify regression and other estimation models in the `command()` option, we have some keywords we can use to request returned results.
- Type `help table` to see the full list.
- We will see how `etable` can make this easier later.



# Formatting and style

- There are many more formatting and style options in the collect suite, but let's start with the basic options available in table:
  - `nformat(%fmt [results])`: specify numeric format
  - `sformat(sfmt [results])`: specify string format
  - `cidelimiter(char)`: use *char* as delimiter for confidence interval limits
  - `cridelimiter(char)`: use *char* as delimiter for credible interval limits
  - `stars(starspec)`: add stars to denote statistical significance
  - `style()`: use a predefined style or a style file created using collect
  - `label()`: use label file created using collect

# Formats

- If we were reporting summary statistics, we would specify the summary statistics within the formatting options.

```
. table union collgrad, statistic(frequency) ///  
statistic(percent) nformat(%2.0f percent) ///  
sformat("%s%%" percent)
```

- Because % is a special character, we added two of them to the string format.

# Export

- The `export()` option was added to `table` in Stata 19.

```
. table union collgrad, statistic(frequency) ///  
statistic(percent) nformat(%2.0f percent) ///  
sformat("%s%%" percent) export(tabulation.docx)
```

- Supported file extensions include `.docx`, `.html`, `.pdf`, `.xls`, `.xlsx`, `.tex`, and others.

## Creating *Table 1* with the dtable command

# The dtable command

- While `table` can be used to create tables of summary statistics, the `dtable` command (introduced in Stata 18) can make this task a bit easier. It is also built on the `collect` system, but it has many default options that are specific to creating descriptive tables, usually known as Table 1.

# Descriptive statistics

- Often, we need to report descriptive (summary) statistics for variables used in an analysis. For continuous variables, we might consider reporting their means and standard deviations. This goal can be achieved by the `table` command we saw before.

```
. table () result, statistic(mean wage hours tenure) ///  
statistic(sd wage hours tenure)
```

- We can make this table more easily using `dtable`.

```
. dtable wage hours tenure
```

# Descriptive statistics

- We might also have categorical variables and would like to report their frequencies and percentages. This can be easily done by using `dtable`, and we just need to prefix the categorical variables using factor-variable notation.

```
. dtable wage hours tenure i.south i.smsa
```

- Alternative syntax:

```
. dtable, continuous(wage hours tenure) factor(south smsa)
```

# Descriptive statistics

- The feature of supplying different types of variables enables the possibility for handling different reported statistics.

- Alternative syntax:

```
. dtable, continuous(wage hours tenure, ///  
statistics(mean sd)) factor(south smsa, ///  
statistics(fvfrequency fvpercent))  
  
. dtable, factor(south smsa, statistics(fvproportion)) ///  
continuous(wage hours tenure, statistics(min median))
```

- By default, dtable reports the mean and standard deviation for continuous variables and frequencies and percentages for categorical variables.



# Descriptive statistics

- You can specify descriptive statistics in separate options, which is

```
. dtable, continuous(wage hours, statistic(mean)) ///  
continuous(tenure, statistic(iqr)) ///  
factor(south, statistic(fvfrequency)) ///  
factor(smsa, statistic(fvpercent)) ///  
note(Mean reported for wage and hours) ///  
note(Interquartile range reported for tenure) ///  
note(Frequency reported for south) ///  
note(Percent reported for smsa)
```

- We can also denote which statistic is used in note().
- See help dtable for the complete list of statistics supported.

# Across groups and hypothesis tests

- Usually, we might want to describe variables across samples.
- Let's see how we can create such a table using dtable by adding the by() option.

```
. dtable wage hours tenure i.south i.smsa, by(union)
```

## Across groups and hypothesis tests

- Now we get descriptive statistics for the union workers, nonunion workers, and total samples. If we would like to suppress the column for total samples, we add the `nototals` suboption in the `by` option.

```
. dtable wage hours tenure i.south i.smsa, by(union, nototals)
```

- An important feature of this `by` option is that we can perform tests of equality between samples.

```
. dtable wage hours tenure i.south i.smsa, by(union, nototals tests)
```

# Group comparison tests

- By default, dtable uses regress for comparing the difference between groups for continuous variables (note that this is equivalent to ttest for binary groups) and Pearson's  $\chi^2$  test for categorical variables.
- Type `help dtable` to see the complete list of test techniques supported.

# Across groups and hypothesis tests

- To specify different types of tests, we need to utilize the alternative syntax supplying the categorical and continuous variables separately. First, we can try the suppressing test for some variables.

```
. dtable wage hours tenure i.south i.smsa, ///  
by(union, nototals tests) continuous(tenure, test(none)) ///  
factor(smsa, test(none))
```

- Or use Fisher's exact test for all categorical variables.

```
. dtable wage hours tenure i.south i.smsa, ///  
by(union, nototals tests) factor(, test(fisher))
```

# Style

- dtable has several style options, including controlling the display of sample size, changing the column headings, adding titles and notes, changing numeric and string formatting, and specifying style and label files.
- Let's revisit the last descriptive table we created. By default, the sample size is included in the first row. We can use the `sample()` option with its suboption `place()` to specify where we would like to put the sample size. The three supported locations are `items`, `inlabels`, and `seplabels`.

```
. dtable wage hours tenure i.south i.smsa, ///  
by(union, nototals tests) ///  
sample(, statistic(frequency) place(seplabels))
```

# Style

- We can control the look of the sample size by using the `sformat` option.

```
. dtable wage hours tenure i.south i.smsa, ///  
by(union, nototals tests) ///  
sample(, statistic(frequency) place(seplabels)) ///  
sformat("(N=%s)" frequency)
```

# Style

- Finally, we can give the columns new headers, display only two digits past the decimal point of the  $p$ -values, and add a title and note to the table.

```
. dtable wage hours tenure i.south i.smsa, ///  
by(union, nototals tests) ///  
sample(, statistic(freq) place(seplabels)) ///  
sformat("(N=%s") frequency) ///  
column(by(hide) test(P-value)) nformat(%5.2f _dtable_test)///  
title(Table 1. Descriptive Statistics) ///  
note(Mean (standard deviation) reported for wage hours tenure) ///  
note(Frequency (percent) reported for south and smsa)
```



# Export your table

- The `export()` option is also available, just like with `table`.

## Creating estimation tables with the `etable` command

# etable

- While `table` can be used to create tables of estimation results, the `etable` command can make this task a bit easier. It is also built on the `collect` system, but it has many default options that are specific to creating estimation tables.

# Adding results

- Let's start with a regression model.

```
. regress wage hours tenure
```

- We can make a table of these results by simply typing etable.

```
. etable
```

- Add  $R^2$  to our table

```
. etable, mstat(r2)
```

- Two mstat() calls

```
. etable, mstat(r2) mstat(N)
```

- Let's say we wanted a table of coefficients and confidence intervals rather than coefficients and standard errors.

```
. etable, mstat(r2) mstat(N) cstat(_r_b) cstat(_r_ci)
```

## Adding and formatting results

- Each result is specified separately to allow different labels and formatting options for each.

```
. etable, mstat(r2) mstat(N, label("Sample size")) ///  
cstat(_r_b) cstat(_r_ci, cidelimiter(,) nformat(%4.2f))
```

- Once we have options we're happy with, we don't have to repeat them each time we make changes to our table. Rather, we use the `replay` option to continue to use the results and styles we've already set.

```
. etable, replay
```

# Adding and formatting results

- We can add results from additional models.

```
. regress wage hours south##smsa  
  
. etable, append  
  
. regress wage hours tenure south##smsa i.union  
  
. etable, append  
  
. etable, keep(hours tenure) replay
```

- To see all our coefficients again, we use either `*` or `_all` in `keep()`.

```
. etable, keep(*) replay
```

# Style

- `etable` has several style options, including adding stars, changing the column and row headings, adding titles and notes, and specifying a style and label file. To start, we may want to make the column headings more descriptive. By default, `etable` uses the dependent variable as the column heading. We can change the column headings with the `column()` option.
- See `help etable`.

# Style

- In our case, all our models have the same dependent variable, so we may just want to label the columns as models 1, 2, and 3. We can do this with option `column(index)`.

```
. etable, replay column(index)
```

- To add stars of significance, we can simply add the `showstars` option and the `showstarsnote` option to add a note explaining the stars.

```
. etable, replay showstars showstarsnote
```



# Style

- By default, etable will add one star (\*) to the coefficient if  $p < 0.05$  and two stars (\*\*) if  $p < 0.01$ .

```
. etable, replay stars(.05 * .01 ** .001 ***, prefix(Note:))
```

- Let's also add the center option to center our result columns.

```
. etable, replay center
```

- Finally, we can add a title and export our table to an Excel spreadsheet.

```
. etable, replay title(Table 2. Regression results.) export(wage.xlsx)
```

# Equation options

- There are three equation options you may need to create the table you want using etable: `equations()`, `showeq`, and `eqrcode()`.
- Let's return to the multivariate regression model we created earlier, this time creating it with etable.

```
. mvreg wage hours = i.union i.collgrad  
  
. etable
```

# Equation options

- The results from both outcomes are in one column because they come from the same model, but we can't tell which are which. Adding the `showeq` option will add labels for each set of results.

```
. etable, showeq
```

- If we want to see results from the wage equation, we can add `equation(wage)`.

```
. etable, equation(wage)
```

# Equation options

- Let's compare this table with one in which we fit two separate linear regression models: one on wage and one on hours.

```
. regress wage i.union i.collgrad  
  
. estimates store Wage  
  
. regress hours i.union i.collgrad  
  
. estimates store Hours  
  
. etable, estimates(Wage Hours)
```

- Because our two outcomes come from two different models, they are now in separate columns, but the results are still stacked as they were before. We can see this clearly if we add the `showeq` option once more.

```
. etable, replay showeq
```

# Equation options

- To get the results from both models into the same rows, we need to give them both the same equation name (i.e., label the wage results as hours results). We can do this with the `eqrcode()` option.

```
. etable, replay eqrcode(hours=wage) noshoweq
```

- We can also tell `etable` to label our columns using the names we gave when we stored our results.

```
. etable, replay column(estimates)
```

- Finally, we can export our table to a Word document.

```
. etable, replay export(regresults.docx)
```

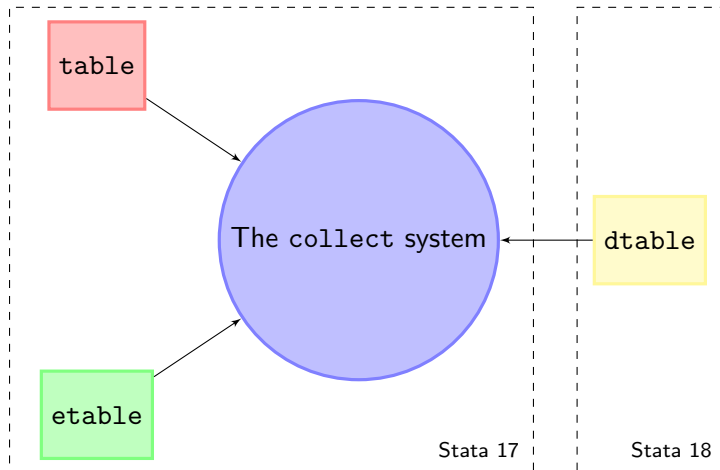
## The collect system

# Tables are collections

- `table`, `etable`, and `dtable` automatically create collections that can then be modified and exported with the `collect` suite. Collections consist of results stored by commands and their corresponding labels, formats, and styles.

- └ The collect system
- └ Tables are collections

## Stata's table framework





# Tables are collections

- Let's start with the first table we made in this course. First, we'll need to clear all collections in memory.

```
. collect clear
```

```
. table union
```

- Behind the scenes, this command created a collection called Table. We can see a list of the current collections using the collect (or collect dir) command.

```
. collect
```

- └ The collect system
- └ Tables are collections

# Tables are collections

- The way items are tagged is with dimensions and levels of those dimensions. You can see the dimensions the items in our collection are tagged with in `collect dims`.

```
. collect dims
```

- We can see the levels of any dimension with the command `collect levelsof`.

```
. collect levelsof cmdset
```

```
. collect levelsof command
```

```
. collect levelsof result
```

```
. collect levelsof statcmd
```

```
. collect levelsof union
```

- └ The collect system
- └ Tables are collections

## Tables are collections

- You can see the associated labels of of these levels using `collect label list`.

```
. collect label list union
```

- All in all, you can think of each item in our table being tagged in the following way:

Item	Tags
1,417	union[ 0] result[frequency] statcmd[1]
461	union[ 1] result[frequency] statcmd[1]
1,878	union[.m] result[frequency] statcmd[1]

- To see how these tags were used to create a table layout, we can type `collect layout` or `collect query layout` in Stata 19.

```
. collect layout
```

```
. collect query layout
```

- └ The collect system
- └ Tables are collections

# Tables are collections

<div>union</div> <div>[.m] [1] [0]</div>	<div>result</div> <div>[frequency]</div>	
		Frequency
	Union worker	
	Nonunion	1,417
	Union	461
Total	1,878	

## General flow

- The general workflow of creating a table with `collect` looks like this:
  - 1 Collect results from Stata commands.
  - 2 Lay out the rows and columns of your table.
  - 3 Customize your table—formats, labels, style, bolding, italics, colors, and more. You can also specify style and label sets that you have previously saved.
  - 4 Export your table to Microsoft Word, Excel, PDF,  $\text{\LaTeX}$ , and more.
  - 5 Save your style, labels, or whole collection to use and modify in the future.
- Along the way, we will also use commands to see what's in our collection, create advanced customizations, and manage our collections.

# Collect results

- There are two other ways to collect results from commands:  
collect: and collect get.
- We don't have any collections we would like to hold on to, so we'll clear all collections.

```
. collect clear
```

## collect [get]: prefix

- To start our collection, we can use `collect [get]:` as a prefix, before any command and it will collect everything that is returned by that command in `e()` or `r()`. The word `get` is optional.

```
. collect clear
```

```
. collect: regress wage hours tenure
```

- By default, the `collect` commands create a collection called `default` unless the `name()` option is used.

```
. collect
```

- Once again, we can use `collect dims` to see what's in our collection.

```
. collect dims
```

## collect: prefix

- Of interest, we have 3 levels in `colname` and 33 levels in `result`.  
Let's see what's in `colname` using `collect label list`.

```
. collect label list colname, all
```

- Let's see what's in `result`.

```
. collect levelsof result
```

- We can see that our collection contains everything returned by `regress`.



# collect get

- The other way we could have collected results is by first running our command and then running `collect get` afterward. Let's specify a slightly different model this time.

```
. regress wage hours tenure i.union
```

- We added union membership as a main effect. When we collect these results, they will be added to the current collection.

```
. collect get _r_b _r_se
```

```
. collect
```

- Let's collect results from one more model (the third one!).

```
. regress wage hours c.tenure##union
```

```
. collect get _r_b _r_se
```

# collect get

- Now let's see how the tags in our collection have changed.

```
. collect dims
```

- We can see that `cmdset` now has three levels, and the number of `colname` levels has increased. Let's see what they are.

```
. collect levelsof colname
```

- The factor variables included in the model (in our case, just `union`) are now dimensions as well.

- └ The collect system
  - └ Lay out results in a table

## Lay out results in a table

- We can create table layouts using `collect layout`. Its row, column, and table specifications work largely the same as `table`, but there are a couple of difference that we will discuss below.  
`collect layout (rowdims) (coldims) (tabdims)`

- └ The collect system
  - └ Lay out results in a table

## Explicit specification

- We would like to make a table out of these collections.
- First, we would like to make a table using `colname` and `result`.

```
. collect layout (colname) (result)
```

- We have only two results being displayed in the table.

- └ The collect system
  - └ Lay out results in a table

## Explicit specification

- Only coefficients and standard errors are shown because that's what we specified that we wanted to see when we collected these results.
- In order to show all results from all models, we need to add `cmdset` to our layout.

```
. collect layout (colname) (result) (cmdset)
```

- Unlike `table`, however, we can't just list all our row specifiers.

```
. collect layout (colname result) (cmdset)
```

- We will need to interact levels.

```
. collect layout (colname#result) (cmdset)
```

## Customize your table

- For this section, we would like to change the look of our table by modifying the labels and style and adding stars signifying statistical significance. Let's start with labels.

# Labels

- Just as with `table`, we can change the variable and value labels in our dataset to change the headers shown in our table. With `collect`, however, we have another option: we can directly change the labels and headers in our collection and save them to use in future collections.
- Here are the commands available to adjust labels:
  - Add or modify the label for a dimension.  
`collect label dim dim "label" [, modify]`
  - Add, modify, or replace labels for levels within a dimension.  
`collect label levels dim level1 "label" [level2 "label" ...] [, modify replace]`

- └ The collect system
- └ Customize your table

# Labels

- In our table, instead of listing the models as 1, 2, and 3, we would like them to say “Model 1”, “Model 2”, and “Model 3”. This would be an example of changing the level labels of the dimension `cmdset`.

```
. collect label levels cmdset 1 "Model 1" 2 "Model 2" 3 "Model 3"
```

- Let's also take the “(years)” out of the label for `tenure` and change the label of `hours` to just “Hours”.

```
collect label levels colname tenure "Job tenure" hours "Hours", modify
```

- In this case, we need to add the `modify` option because there are existing labels for `colname`. If we had used the `replace` option, all the other labels for `colname` would have been deleted, and only the new labels for `tenure` and `hours` would have remained.



## Formatting results

- We can change numeric and string formatting using the same syntax we used with `table`; we just need to specify which cells we would like it applied to.
- For our table, let's say we want most results to have two decimal places and  $p$ -values to have three decimal places. Furthermore, we would like to put standard errors in parentheses.

```
. collect style cell result[_r_b _r_se], nformat(%6.2f)
```

```
. collect style cell result[_r_se], sformat("(%s")
```

## Adding stars

- Finally, we'd like to add stars of significance, a title, and some notes to our table. First, let's add stars to indicate significance. We'll use the same scheme we used previously.

```
. collect stars _r_p 0.05 "*" 0.01 "***" 0.001 "****", ///  
attach(_r_b) shownote
```

- By adding the shownote option, we get a note at the bottom of our table explaining our stars scheme.

## Final touches

- We can add additional notes using `collect notes`. For example, we may want to add information about the source of this dataset. You can add a note to a specific position (first, second, third note) by adding the number before a colon. Without a position specification, notes are added below existing notes.

```
. collect notes "1988 data, extracted from National Longitudinal of  
Young Woman who were ages 14-24 in 1968 (NLSW)."
```

- Finally, we can add a title to our table.

```
. collect title "Table 2. Regression results"
```

# Final table

- We can use `collect preview` to see our updated table (`collect layout` works too).

```
. collect preview
```

- Use `collect export` to export the table.

```
. collect export table2.pdf
```

## Reproducible report

# Intro

- So far, we have learned how to create customized tables and export them, but what if we wanted to incorporate our table into a larger report?
- There are two varieties of commands for creating reports in Stata.
  - The first variety creates Word documents, PDF documents, and Excel files that incorporate stored results from Stata commands in formatted text and tables. The `put` suite of commands creates documents in this manner.
  - The second variety creates HTML and Word documents that include the full output from Stata commands and allows you to format the text using Markdown. The `dyn` suite of commands incorporates Stata output in this manner.

# Intro

- The `putdocx` suite of commands offers the most functionality. Therefore, even if you're interested in creating a PDF, you may consider using `putdocx` instead of `putpdf` and then converting the resulting Word document to a PDF using the `docx2pdf` command.
- The syntax of `putpdf` is very similar to `putdocx` but has some different options.

# Introducing putdocx

- The basic commands to create a Word document in Stata are as follows:
- Create, save, and append Word files (see [\[RPT\] putdocx begin](#))
- Insert page breaks in a Word file (see [\[RPT\] putdocx pagebreak](#))
- Add paragraphs with text and images (see [\[RPT\] putdocx paragraph](#))
- Add tables to a Word file (see [\[RPT\] putdocx table](#))



# Create a complete report in Stata

- Let's create a simple report of results using putdocx. All the commands we need are in putdocx.do.
- Creating a PDF document in Stata works largely the same as creating a Word document, with some different options. Remember that you can also always convert a Word document into a PDF using the docx2pdf command. You don't even need to have Word installed to do this. For example,

```
. docx2pdf putdocx.docx, replace
```

- To see a complete example of creating a PDF document, see putpdf.do.

## Creating an Excel file

- The idea behind `putexcel` is similar to `putdocx` and `putpdf` in that we would like to write text, returned results, images, and tables into an Excel file. We'll see, however, that the functionality is quite different. It's more similar to `putdocx table` or `putpdf table` in that we specify the cell in the Excel spreadsheet that we would like to insert our content into. We can either create an entire Excel spreadsheet from within Stata or we can write content to specific cells of an existing spreadsheet by adding the `modify` option to `putexcel set`.

## Creating an Excel file

- Four basic commands that you need to create an Excel file in Stata:
  - `putexcel set filename`
  - `putexcel ul_cell = content`
  - `putexcel cellrange, formatting_options`
  - `putexcel save`
- When we were creating Word documents, we began our document in Stata and only saved it to a file at the end. When we create Excel files, we start by creating a blank file, and every command we issue will update that file as we go. Hence, we use `set filename` instead of `begin` to start our document. At the end, `putexcel save` will save and close the file.

## Creating an Excel file

- To add content to our Excel file, we specify the upper-left cell (*ul\_cell*), where the content will begin. We can add formatting when we add content, or we can specify a cell range (*cellrange*) to change the formatting later. Specifying a cell range often involves less typing than specifying a format for each individual cell.
- Excel 1997/2003 (.xls) files and Excel 2007/2010 and newer (.xlsx) files are supported.

# Example report

- All the commands used to make this report are in `putexcel.do`.  
Let's run it now.

```
. do putexcel.do
```

# Creating dynamic documents

- You can also create HTML reports using the Markdown text-formatting language. There are three dynamic documenting commands.

<code>dyndoc</code>	Convert dynamic Markdown document to HTML or Word
<code>dyntext</code>	Process Stata dynamic tags in text file
<code>markdown</code>	Convert Markdown document to HTML file or Word
- Dynamic tags are instructions used by `dyndoc` and `dyntext` to perform a certain action, such as running a block of Stata code, inserting the result of a Stata expression in text, exporting a Stata graph to an image file, or including a link to the image file.

# Markdown to HTML

- `dyndoc` converts a dynamic Markdown document—a document containing both formatted text and Stata commands—to an HTML file or Word document. Markdown is a simple markup language with a formatting syntax based on plain text.
- Perhaps the easiest way to learn Markdown and dynamic tags is to look at a Markdown file and the resulting HTML file side-by-side. You can find the Markdown file in your directory. It's called `dyndoc_ex.txt`. We can create the HTML file from the Markdown file using `dyndoc`.

# Markdown to HTML

- The basic syntax for `dyndoc` is as follows:

`dyndoc srcfile [arguments] [, options]`

- *srcfile* is a plain text file containing Markdown-formatted text and Stata dynamic tags.
  - The options we will use are `saving()` and `replace`.
- You can also write Markdown-formatted text to a Word file by adding the `docx` option. Or you can convert HTML documents to Word documents using the `html2docx` command.

```
. dyndoc dyndoc_ex.txt, replace
```



# Markdown to HTML

- We could also convert our HTML file to a Word document.
- Or we could have created a Word document directly with `dyndoc`.

```
. dyndoc dyndoc_ex.txt, docx replace
```

- We won't go over them, but `dyntext` and `markdown` work largely the same. If you want to convert a Markdown document without Stata dynamic tags to an HTML file or Word document, see [\[RPT\] markdown](#). If you want to convert a plain text file containing Stata dynamic tags to a plain text output file, see [\[RPT\] dyntext](#). Both of these can be thought of as special cases of `dyndoc`.

**Thank you!**