# STATS 22: Stata Workflow and Tips

Course Notes

Peter Liu

April 2, 2020

## Contents

# 1 Introduction

Upon opening the Stata software, there are several features of the window to note. In the main **Results** section of the window, the results of any commands executed will be displayed. Directly below is the **Command** section, where Stata commands can be entered. To better demonstrate window features, we can load an example dataset built into the software, by going to *File > Example datasets...* and clicking "Example datasets installed with Stata". Then, we may use the `bpwide.dta` dataset by pressing the "use" option by its name.

Upon doing so, we notice that a command has been run in **Results**: `sysuse bpwide.dta`. This command was used to load the `bpwide.dta` dataset into Stata, and we can also see that the dataset describes "fictional blood-pressure data". The **Variables** section on the top-right of the window shows what variables are in the dataset we just loaded, and if we click each individual variable, the **Properties** section on the bottom-right will show the name, label, type, and other relevant information about that particular variable.

We can also run some example commands in the **Command** section.

- `describe` will display information about all variables in the dataset, similar to how **Properties** describe individual variables
- `summarize` will display summary statistics for each variable, such as the minimum, maximum, number of observations, etc.

If you want a visual grasp of what the dataset contains, click *View > Data Editor > Browse* from the menu bar. This will show a table with columns for the variables and rows for the observations.

## 1.1 Stata Documentation

In learning Stata's syntax, an invaluable command to enter is the `help` command. Entering this by itself will yield a popup window with some advice on finding help with Stata, but even more helpful is entering `help` followed by some command you want to learn more about. For example, `help summarize` will show you the documentation for the `summarize` command, which includes the syntax.

**Syntax of `summarize`:**

```
summarize [varlist] [if] [in] [weight] [, options]
```

The help file here tells us that `summarize` can be followed by `[varlist]`, which is a list of variables that you would like summarized, `[if]` conditions that specify which observations to summarize, and more. For example, if we only want to summarize the `patient` and `sex` variables in the loaded dataset, run the command:

```
summarize patient sex
```

If we only want to summarize the variable `patient` for males only, we can apply the `[if]` command option.

```
summarize patient if sex == 0
```

We know that males have the `sex` variable being `0`, because in the Data Editor (Browse) window, clicking a "Male" observation in the `sex` column will show in the top bar that the value is stored as `0`.

We can also see from the help file of `summarize` that the last parameter is `[, options]`. In reality, this parameter encompasses several more options that the summarize command can be followed by, and the table directly below the syntax line in the help file explains to you what those options are. Note that the word `options` is preceded by a comma, meaning if you want to use those options as part of `summarize`, you must write a comma first. For example, if you enter `summarize, detail` as a command, Stata will "display additional statistics", according to the help file. Scrolling down to the **Options** section of the help file, we can get even more information about the `detail` option (and other options).

Note that the help file also has an **Examples** section demonstrating the command in actual use. If you want even further detailed documentation, it's possible to open a PDF of the complete Stata documentation by clicking "View complete PDF manual entry" at the top of any command's help file. For example, doing so from the `summarize` help file will bring you to the `summarize` command section of the Stata

documentation PDF. Scrolling down a bit will even show a **Methods and formulas** section that explain the mathematical/statistical methods internally used by Stata to compute results given by `summarize`.

## 1.2 Do-Files

If you restart Stata, you'll find that all the previous commands' results have been lost. The way to save commands in a Stata "script" is with a **do-file**. A do-file allows you to write a series of commands in a file, and save this document on your computer. Then, when you open Stata, this do-file can be run, and all the commands entered will be replicated. To create a do-file, click on *File > New > Do-file*. After writing any commands in this file, you can click the *Execute* button on the top-right to execute those commands. Returning to the **Results** section of Stata, you'll see that all those commands have been run, and a message that tells you the do-file has finished execution will be displayed:

```
.
end of do-file
```

To run a pre-existing do file, go to *File > Do...* and select the required `.do` file.

## 1.3 Additional Tips

### Command Abbreviations

Stata recognizes **command abbreviations** like `summ` for `summarize`, or `descr` for `describe`. Instead of typing the full command like `describe`, to save time you may type just a few letters of the word, and Stata will still recognize that you are trying to run the `describe` command. Even if you just type `d`, Stata will recognize this as the `describe` command, because there is no ambiguity in what the abbreviation stands for.

However, if there is ambiguity, like just entering `s` in attempting to run `summarize`, Stata will display an error message:

```
. s
command s is unrecognized
r(199);
```

This abbreviations technique will work not only for commands, but also variables, as we'll see later on.

### Automatic Recognition of Typing

As long as you're in the Stata window, even if your mouse selection is somewhere outside the **Command** section, typing on your keyboard will automatically be recognized by Stata as your attempt to use the **Command** section. That is, you could have just clicked into the **Variables** section of the window to examine variable properties; without re-clicking the **Command** section, as soon as you start to type, Stata will bring you back to the **Command* section, ready for you to press enter and execute what you just typed. Try this out yourself!

# 2 Input and Output

To open your own dataset (`.dta` file) in Stata, click on *File > Open...* and select the .dta file of your own choosing. As you can see, the variables from the selected dataset will be loaded into the **Variables** section, replacing any pre-existing variables. To clear out a dataset from Stata's memory, enter the `clear` command, and all variables will be emptied out.

## 2.1 Memory Allocation

The `memory` command will show you how many bytes of memory Stata is currently using, and how much Stata has allocated for you to work with. If you've cleared out any datasets, you'll see that the amount of memory that Stata has `used` is 0, but opening a dataset and then running this command will display a positive number here.

Stata will perform **automatic memory allocation**, which means if you load a dataset of a size greater than the amount of memory allocated, Stata will automatically allocate more memory. In practice, you'll never need to worry about memory, unless you've reached the limit set by your computer hardware. If you have reason to believe that you need to use less memory for the same dataset, you can use the `compress` command. Stata will go through each variable in your dataset and evaluate whether there is some form of internal storage that takes up less memory. It will transform each variable to a less memory-intensive storage type without compromising essential information.

## 2.2 Working Directory

Say you have a data file called `temperatures.dta` located on the desktop of your computer. You'd like to open this file, so you enter the following command, mimicking how Stata opens datasets when you use *File > Open...*.

```
use "temperatures.dta"
```

If your **present working directory** is not the desktop, Stata will return an error. This is because Stata will only be able to locate files by name within your working directory, as it is not smart enough to look elsewhere on your computer. To check your present working directory, use the command `pwd`.

To change your working directory, go to *File > Change working directory...* and select the directory in which your `.dta` file is located. Now, directly entering `use "temperatures.dta"` will succeed.

## 2.3 Exporting

Once you've loaded a `.dta` dataset, you may want to export it into another format, such as a **comma-separated values** file (`.csv`) or **Excel** file (`.xls` or `.xlsx`). Stata has such capabailities in *File > Export* and you can select the desired option for the file format. As an example, if you click the option **Data to Excel spreadsheet (*.xls;*.xlsx)**, a menu will pop up prompting you to specify the particular variables you'd like exported (or leave blank to include all variables), and to specify the filename, among other options. To specify the export name and destination, you can use the *Save as...* button. Just press OK and you can see the command run by Stata to perform the export.

## 2.4 Importing

To import a dataset that is not necessarily a `.dta` file format, click *File > Import* and select the target file's format. As an example, select the text data option, and click the folder icon to select the text file. Be sure to manipulate the available settings in this window to correctly import your dataset. For example, if the first row of your text file are variable names, correctly specify this with the *First row as variable names* option. If your are importing a `.csv` file, specify the *Delimiter* option to be "Comma". Although, the "Automatic" option usually works well.

Remember that you can use a pre-installed dataset by going to *File > Example datasets...*, and clicking the link "Example datasets installed with Stata". Then, press "use" for whatever example dataset you'd like. You may also follow the link to "Stata 16 manual datasets", which contains all datasets to which the Stata documentation refers.

Datasets which you opened recently can be accessed from the *File > Open recent* menu.

# 3 Edit Options

This section will cover some of the options available under *Edit* in the menu bar.

## 3.1 Clearing Results

To clear the **Results** section of the Stata window, you must first select the **Results** section, then go to *Edit > Clear results*. This is not the same thing as entering `clear` as a command! The `clear` command will clear Stata's memory and remove any variables previously loaded. Clearing results will only visually empty the **Results** section of the window.

## 3.2 Find

The need to first select the **Results** section is also evident for the options in *Edit > Find*. The *find* options will not be available if you have selected another section.

## 3.3 Copy

When using copy and paste for text in the **Results** section, you can take advantage of the *Edit > Copy table* option. For example, if you `summarize` a dataset, and would like to copy the resulting table into Excel, simply doing Command/Control-C and Command/Control-V will put all table contents into one Excel column. Instead, if you select `summarize`'s resulting table and click *Edit > Copy table*, pasting into Excel now will nicely format the table into proper cells.

*Edit > Copy as HTML* will yield a similar result, but with more stylized features like borders. *Edit > Copy as picture* will copy the entire table as an image.

# 4 Dataset Management

In this section, we'll begin the core work involved with a dataset, including inspection and modification. This is the first step towards true data analysis using Stata.

## 4.1 Describing Data

In describing a dataset, an option is to go to *Data > Describe data > Describe data in memory or in a file.* A set of data in memory is data that you've already loaded into Stata using the `use` command, either explicitly typing this or by opening a dataset from the menu bar. A dataset in a file is a `.dta` file that has not yet been loaded. If you describe data in memory and include all variables (with the default selection of options), this is the same as directly executing the `describe` command.

As you can typically see from the result of `describe`, it is generally good practice to leave the variable names to be short, while allowing the variable label to be longer and more descriptive.

*Data > Describe data > Describe data contents (codebook)* will generate more detailed descriptions for each variable in the dataset. This includes the variable storage type, value range (for numerics), number of unique values, number of missing values (denoted by a `.` in the observation), percentiles (for numerics), and more. The equivalent command for this menu bar selection is `codebook`.

*Data > Describe data > Inspect variables* will generate some rough graphs that represent the distribution of each variable, but they are rough because in reality, they're written with text. Try this for yourself and see. There is also information about the number of negative, zero, and positive observations for each numeric variable. The equivalent command is `inspect`.

*Data > Describe data > Compactly list variable names* is self explanatory, and the equivalent command is `ds`.

*Data > Describe data > Summary statistics* is exactly the `summarize` command we've been using.

## 4.2 Viewing, Editing, and Labeling a Dataset

As previously mentioned going to *Data > Data Editor (Browse)* will allow you to see your dataset in a table format. However, if you try to change any value, you will be unable to do so. This is because you need to enter the "Edit mode" on the top-left first. Making any edits will be reflected in the **Results** section of the window, showing you that a command was executed to replace some data.

To label a dataset, go to *Data > Data utilities > Label utilities > Label dataset.* Labelling the dataset is like giving your dataset a title. Stata will execute the command `label data "label name"`.

## 4.3 Listing Data and `if`-Statements

Data can be listed by gong to *Data > Describe data > List data*, and leaving the variables field empty will be equivalent to running the command `list`. This one way to browse data, alternative to using the Data Editor. However, the advantage of listing data prevails when combined with an `if` statement. Going to the same menu bar selection of *List data*, clicking the "by/if/in" tab will allow you to enter an `if` statement, which narrows down the data. Only data satisfying those conditions will be listed.

For example, if we open the `auto2.dta`. example dataset, we can enter in the "If: (expression)" field:

```
price > 10000
```

Then, clicking OK will cause Stata to list only the observations of cars whose price variables takes a value greater than `10000`. As you can see, the equivalent command would be `list if price > 10000`.

As another example, we can enter the following command to list only the data whose miles per gallon (`mpg`) variable is greater than the mean, `21.2973`. The mean of this variable can be obtained by the command `summarize`.

```
list if mpg > 21.2973
```

However, note that if your condition is whether a variable is *equal* to some value, you must use the double equal sign (==). For example, if we want to list the observations in which `mpg` is exactly 20, run the command:

```
list if mpg == 20
```

You can also list only specified variables that meet a certain condition by naming those variables before the `if` word.

```
list make mpg if mpg == 20
```

## 4.4 Appending Datasets

Appending to a dataset means adding more observations to the bottom of the dataset. If the appended dataset contains variables with the same name as the existing variables, then they will be appended to the corresponding columns. If the appended dataset contains new variables, then new columns will be created and used. To append a dataset to another dataset you've already loaded, go to *Data > Combine datasets > Append datasets*. This will bring up a menu that allows you to choose a `.dta` file to append. The equivalent command is `append using ...`, where `...` is the file name/path of the appended dataset.

As mentioned, if the appended dataset has a new variable, then a new column will be created. Stata will leave missing values (`.`) for the original observations that did not have that variable, as well as for any new observations that do not have the original variables.

## 4.5 Merging Datasets

While similar to appending datasets, merging is instead finding a common variable between two datasets and combining the datasets' variables by matching observations. Suppose we have the following dataset named `merge1.dta` with ten observations and two variables.

**`merge1.dta`:**

| id | A |
|---|---|
| 1 | 100 |
| 2 | 100 |
| 3 | 100 |
| 4 | 100 |
| 5 | 100 |
| 6 | 100 |
| 7 | 100 |
| 8 | 100 |
| 9 | 100 |
| 10 | 100 |

Suppose we have another dataset called `merge2.dta` with five observations and two variables, including the same variable `id` as in `merge1.dta`.

**`merge2.dta`:**

| id | B |
|---|---|
| 1 | 200 |
| 2 | 200 |
| 3 | 200 |
| 4 | 200 |
| 5 | 200 |

If we have loaded `merge1.dta`, and we would like to merge `merge2.dta`, we can go to *Data > Combine datasets > Merge two datasets*, and select the desired merge target on disk. However, simply pressing OK here will yield an error, because we also need to specify what the "Key variables" are for Stata to look at. The key

variables will indicate the common variable between the datasets that Stata will use to match observations and add the new variable. In our case, the key variable is `id`, as we can select from the drop-down menu.

Pressing OK will generate the following resulting dataset:

| id | A | B | _merge |
|---|---|---|---|
| 1 | 100 | 200 | matched(3) |
| 2 | 100 | 200 | matched(3) |
| 3 | 100 | 200 | matched(3) |
| 4 | 100 | 200 | matched(3) |
| 5 | 100 | 200 | matched(3) |
| 6 | 100 | . | master only (1) |
| 7 | 100 | . | master only (1) |
| 8 | 100 | . | master only (1) |
| 9 | 100 | . | master only (1) |
| 10 | 100 | . | master only (1) |

What Stata has done is use the `id` variable for each observation to detect commonality between the two datasets. It sees that the `id` of `1` in the first dataset should be matched with the `id` of `1` in the second dataset, so it proceeds to merge the variable `B` to observation number 1. The new `_merge` variable displays the status of the merge, with `matched(3)` indicating a successful merge for that observation, and `master only (1)` indicating that only the *master* (or originally loaded) dataset had something to contribute to the result. For observations 6 through 10, the second dataset did not have those `id`s, so the result only contains the pre-existing information about the variable `A`. In the Stata window, the results will also tell you how many observations were matched, and how many were not matched.

## 4.6 Reshaping Datasets

The main command that will be covered in this section is `reshape`. In the help file for `reshape` (`help reshape`), it is stated that a **long dataset** is characterized by fewer variables whose index variable values are often repeating. For example, the long dataset graphic in the help file has index variables `i` and `j` which both have repeating values, and only one variable `stub` with the actual numerical information. On the other hand, a **wide dataset** has more variables (`stub1` and `stub2`) to represent the actual numerical information, eliminating repeating values in the index variable `i`. Also, the index variable `j` was eliminated altogether.

Suppose we have a long dataset in `long.dta` as follows.

**`long.dta`:**

| year | id | sales |
|---|---|---|
| 2000 | 1 | 100 |
| 2001 | 1 | 100 |
| 2002 | 1 | 100 |
| 2003 | 1 | 100 |
| 2004 | 1 | 100 |
| 2000 | 2 | 150 |
| 2001 | 2 | 150 |
| 2002 | 2 | 150 |
| 2003 | 2 | 150 |
| 2004 | 2 | 150 |
| 2000 | 3 | 200 |
| 2001 | 3 | 200 |
| 2002 | 3 | 200 |
| 2004 | 3 | 200 |

To convert this to a wide dataset, go to *Data > Create or change data > Other variable-transformation commands > Convert data between wide and long.* For the "ID variable" field, place the variable in the original dataset that you want to keep. For example, we decide that we want to keep the `id` variable above,

since the `id` variable is understood to represent Firm #1, Firm #2, and Firm #3. For the "Subobservation identifier", select the variable that you would like dropped. Here, we decide that we don't want a column for `year`, so we place `year` here. For the last drop down menu, select the variable that contains the core information you would like represented in a wide dataset (`sales`).

Pressing OK will yield the following result.

```
id     sales2000     sales2001     sales2002     sales2003     sales2004
1      100           100           100           100           100
2      150           150           150           150           150
3      200           200           200           200           200
```

Originally, we had years for each firm written out extensively in a column, and sales for that particular firm and year were written in the third column. Now, we have only three "observations" (rows), each representing a firm. However, we have not just one sales column anymore. We have a sales column for every single year that was in the original dataset's `year` column.

If we want to go back to the original long form, we can go to *Data > Create or change data > Other variable-transformation commands > Convert data between wide and long*. Then, select the option "Back to long format (previously reshaped)". This time, there's no need to specify parameters, because Stata already has them in memory.

## 4.7 Transposing Datasets

To **transpose** a dataset means to interchange the observations of a dataset to become the new dataset's variables, and to interchange the variables of a dataset to become the new dataset's observations. This is analogous to a matrix transpose, and can be performed in Stata by going to *Data > Create or change data > Other variable-transformation commands > Interchange observations and variables*. Then, acknowledge the warning and press OK. The equivalent command is `xpose, clear`. Transposing a dataset twice will nearly result in the re-formation of the original dataset, but original variable names will be lost. Stata cannot memorize the original variable names when they're transposed into observations.

One fix to this is by selecting the option "Add variable _varname containing original variable names" in the transpose menu. This would cause the transpose to add an additional variable at the very end that contains the original data's variable names. Transposing twice would now retain variable names, but would also add a somewhat useless column at the end denoting "variable names" of the observations.

## 4.8 Comparing Two Datasets

Given the need to compare two similar datasets, or two very large datasets, Stata offers the capability to do so in *Data > Data utilities > Compare two datasets*.

## 4.9 Contracting and Collapsing Datasets

Before going into the `contract` and `collapse` commands, first note that the `sort ...` command will sort a dataset by the hierarchy of variables specified in `....` That is, the first variable written after the word `sort` will be used to sort the observations in ascending order of that variable, and any ties will be broken by subsequent variables specified.

With the `contract` command, Stata will analyze each row of the dataset and see if there are any duplicated rows (e.g. multiple rows that are exactly the same). Stata will count how the frequency of how many times that observation (row) appears, and write this number in a new column. It will then delete the duplicated observations. The `contract` command can be used by going to *Data > Create or change data > Other variable-transformation commands > Make dataset of frequencies*. If you want all variables to be used, select all variables (click one variable at a time).

On the other hand, `collapse` generates a new dataset of descriptive statistics for various combinations of variables. This command can be used by going to *Data > Create or change data > Other variable*

*transformation commands > Make dataset of means, medians, etc.* Then, the desired descriptive statistics can be chosen, such as the mean applied to particular variables. Under the "options" tab, grouping variables can also be specified if you want to calculate the descriptive statistics of observations in groups, such as combinations between city and region.

# 5 Variables Management

This section will cover creating variables, using conditionals to replace variable values, renaming, counting, and other ways of working with variables.

## 5.1 Setting Observations and Variables

Upon opening Stata, no datasets have been loaded, and we can directly create a new dataset of our own. Firstly, we can set the number of observations in our dataset by using the command `set obs n`, where `n` is the number of observations desired.

The `generate` command can be used by going to *Data > Create or change data > Create new variable*. You can specify the variable name and the value it should take. The equivalent command is `generate varname = varvalue`. You can also create another variable that is equal to an existing variable, such as `generate varname2 = varname`.

```
generate var1 = 10
generate var2 = 20
generate var3 = var1
```

A useful way to generate a range variable is by typing `generate var4 = _n`. The `_n` is a special keyword denoting "range", and running this command will generate `var4` to be a sequence of integers from `1` to the highest observation number.

We can replace all variable values that meet a certain condition using the `replace` command followed by an `if`-statement. For example, to replace all instances of when `var3` is equal to `10` with the value `20`, we can write `replace var3 = 20 if var3 == 10`. Note that to specify multiple conditions in the `if` statement, the "or" statement is represented by the `|` symbol, and the "and" statement is represented by the `&` symbol. Here's some sample code replacing values using "or" and "and":

```
replace runiform = 1 if runiform < 0.25 | runiform > 0.75
replace runiform = 2 if runiform >= 0.25 & runiform <= 0.75
```

This code replaces a hypothetical variable called `runiform` which has range $[0, 1]$ with `1` if its value is less than 0.25 or greater than 0.75, and with `2` if its value is in between.

Next, we can reorder variables, such as moving `var3` to the front, by typing `order var3`. The `aorder` command will reorder the variables alphanumerically based on their names.

## 5.2 Variable Labels

Just like for a dataset, variables can have labels too. Variable labels can be checked for existence by the `describe` command. If variables have labels, they should appear in the "variable label" column of table. To label a variable, go to *Data > Data utilities > Label utilities > Label variable*. Select the variable you'd like to label and enter the label. The equivalent command would be `label variable varname "Variable Label"`. This is a very good way to add a description to a variable.

Note that you can also add labels to the contents of a variable. Suppose you have a categorical variable that is internally stored as integers, but you want to add labels to each integer that appears. For example, an `id` variable that stores `1` could represent "Firm 1", `id` storing `2` could represent "Firm 2", etc. We use the same example dataset as above.

**`long.dta`:**

| year | id | sales |
|---|---|---|
| 2000 | 1 | 100 |
| 2001 | 1 | 100 |
| 2002 | 1 | 100 |
| 2003 | 1 | 100 |

```
2004    1    100
2000    2    150
2001    2    150
2002    2    150
2003    2    150
2004    2    150
2000    3    200
2001    3    200
2002    3    200
2004    3    200
```

We can create value labels by going to *Data > Data utilities > Label utilities > Manage value labels*. Press "Create label", give the label a name of your choice (this is not what the label text itself will be), and enter the value and corresponding text you'd like to use as a label for that value. As an example, we can enter 1 as a value, "Firm 1" as its label, 2 as a value, "Firm 3" as its label, and 3 as a value, and "Firm 3" as its label. Pressing OK and going to browse will not display your labels in the `id` column, because we have yet to *apply* the labels.

To apply the labels, go to *Data > Data utilities > Label utilities > Assign value label to variables*. Proceed to choose the variable whose values you'd like labelled, and select the value labels you want. Going to `browse` will show that we've successfully applied the labels.

Note that while these variables have textual representation, they are still numerical variables, because they're stored as integer values. The only difference now is that they have a text label. Using commands like `summarize` will still generate descriptive statistics as if `id` was a numerical variable.

## 5.3 Renaming and Dropping Variables

To rename a variable, type the command `rename var newname`. The variable `var`'s name will be replaced with `newname`.

To drop (or delete) a variable, use the command `drop ...`, where `...` can be a series of variable names separated by spaces to denote the variable you'd like removed. You can also choose to drop all variables *except* certain ones by using `keep A`. This command will drop all variables *except* the variable `A`.

## 5.4 Sorting by Variables

Suppose we have loaded the following dataset:

```
var1    var2
1       100
2       90
3       80
4       70
5       60
6       50
7       40
8       30
9       20
10      10
```

In order to sort the dataset, go to *Data > Sort*, and select a variable to sort. Here, if we choose to select `var2` and press OK, going to `browse` will show that the entire dataset, has been modified, not just the values in `var2`. This is because each observation has a fixed value for `var1` and `var2`, so sorting by `var2` will cause all variables of the observations to be reordered. The equivalent command is `sort var2`.

For a more complex sort, from *Data > Sort* we can select the "Advanced sort" option. If we want to re-sort `var2` in descending order, for example, we can use the syntax `-var2` in the "Variables" field. Note that the

equivalent command for an advanced sort of this kind is `gsort -var2`.

You can also add more variables to the list followed by `sort` or `gsort` in order to specify for Stata what further variables to use in order to break ties with the preceding variables.

## 5.5 Counting Observations

The command `count`, followed by an `if`-statement, will let you know the number of observations that satisfy a certain condition. Or, from the menu bar, you can go to *Data > Data utilities > Count observations satisfying condition*. For example, using the `sysuse sp500.dta, clear` command, we can load the S&P500 example dataset. If we want to count how many observations have the `open` variable exceeding `1300`, we can enter in the "count" window `open > 1300` for the "If" field. This will tell us that `33` observations satisfy this condition. You can see that the equivalent command is `count if open > 1300`.

## 5.6 Operators

The **addition** operator (`+`) can be applied by opening any dataset a generating a new variable by summing two existing variables. For example, with the S&P 500 example dataset, we can execute `gen sum = high + low` to obtain a new variable where each observation's `sum` is the sum of the corresponding `high` and `low` values.

Similarly, we can use the subtraction operator (`-`) by generating a new variable: `gen diff = close - open`.

The **multiplication** operator (`*`): `gen mult = volume * close`. As a side note, numbers too big will be represented in scientific notation in `browse`.

The **division** operator (`/`): `gen div = close / volume`.

The **exponent** operator (`^`): `gen var5 = close^2`.

Note that all the above operators resulted in and are applied to numerical values, since they are arithmetic operators. Stata also has comparison (`<`, `>=`) and logical (e.g. `|`, `&`) operators.

When we used `count if close > open`, we used the "**greater than**" comparison operator `>`. This operator compared the values of `close` and `open` for each observation, and indicated either true or false for each observation's comparison. The usage of the "**less than**" comparison operator `<` is quite self explanatory, but note that for the "**greater than or equal to**" (`>=`) and "**less than or equal to**" (`<=`) operators, the equal sign always follows the comparison sign. This can be remembered by the order of words in the two phrases "greater than or equal to" and "less than or equal to", with the "equal to" portion coming last.

To **check for equality**, we can use the `==` operator. Note that a single equal sign (`=`) is used for assignment, whereas a double equal sign (`==`) is used to check whether two values are equal. For example: `list if high == close`.

To **check for inequality**, use the `!=` operator; `count if open != .` will count how many observations have non-missing values in the `open` variable.

The "**and**" operator (`&`) can be used to combine multiple statements that return true or false, such as `count if open > 1194 & close > 1194`. Statements on both sides of the `&` operator must be true in order for the entire statement "open is greater than 1194 *and* close is greater than 1194" to be true.

The "**or**" operator (`|`) requires than at least one statement on its left or right to be true in order for the entire statement to be true. For example, `count if (open > 1194 | low < 1183)` will count the observation as long as *either* its `open` value is greater than 1194, *or* its `low` value is less than 1183. Also note here the usage of the parentheses, unlike in the above example for `&`. Using parentheses in this example is completely optional, but parentheses can come in very handy when combining multiple logical statements to avoid ambiguity.

The "**not**" operator (`!`) is in fact a logical operator in itself. You can precede any logical statement, such as `open > 1194` with the `!` operator: `!(open > 1194)`. Any observations that returned true for the statement

inside the parentheses will now return false with the preceding `!` operator, and vice versa.

Lastly, displaying the boolean results of a logical statement will output `1` for true and `0` for false. This can be shown by `gen var = (open > close)`, then going to `browse`. For each observation where `open` is greater than `close`, you'll see a value `1` in the `var` column, and `0` otherwise.

## 5.7 Lagged, Forward, Seasonal, and Difference Variables

Suppose we use the previous `long.dta` dataset, with some slight modification to the 'sales column.

```
year     id     sales
2000     1      100
2001     1      110
2002     1      120
2003     1      130
2004     1      140
2000     2      100
2001     2      110
2002     2      120
2003     2      130
2004     2      140
2000     3      110
2001     3      120
2002     3      130
2004     3      140
```

Our eventual goal is create a **lagged variable** for Firm 1's `sales`, so for this example we do not need the data for Firms 2 and 3. We can drop those observations by the command, `drop if id == 2 | id == 3`. Notice that here, `drop` is followed by an `if`-statement that filters out observations satisfying its condition. Thus, `drop` as used here will drop observations rather than variables, which was seen previously. Now, we'll have the dataset:

```
year     id     sales
2000     1      100
2001     1      110
2002     1      120
2003     1      130
2004     1      140
```

Our *desired* dataset is this, with a `lagged_sales` variable that has the `sales` variable "pushed forward" by one observation. Thus, it would have one missing initial value as follows.

```
year     id     sales    lagged_sales
2000     1      100      .
2001     1      110      100
2002     1      120      110
2003     1      130      120
2004     1      140      130
```

For the purpose of this exercise demonstrating lagged variables, we need to convert the data to time-series, by going to *Statistics > Time series > Setup and utilities > Declare dataset to be time-series data*. Select the time variable to be `year` in this case, and select the option "Yearly". Then, run the command `gen lagged_sales = L.sales`. The `L.sales` syntax will lag `sales` by one period. If you want to lag a variable by two periods, run `gen lagged2_sales = L2.sales`. The dataset will then become the following:

```
year     id     sales    lagged_sales    lagged2_sales
2000     1      100      .               .
2001     1      110      100             .
```

| | | | | |
|---|---|---|---|---|
| 2002 | 1 | 120 | 110 | 100 |
| 2003 | 1 | 130 | 120 | 110 |
| 2004 | 1 | 140 | 130 | 120 |

Similarly, we can also create **forward variables** with `gen forward_sales = F.sales`, and we'll have the following dataset. `F.sales` pushes forward the observations in `sales` by one period, so we'll have a resulting missing value in the last observation.

| year | id | sales | lagged_sales | lagged2_sales | forward_sales |
|---|---|---|---|---|---|
| 2000 | 1 | 100 | . | . | 110 |
| 2001 | 1 | 110 | 100 | . | 120 |
| 2002 | 1 | 120 | 110 | 100 | 130 |
| 2003 | 1 | 130 | 120 | 110 | 140 |
| 2004 | 1 | 140 | 130 | 120 | . |

`F2.sales` can also be used to push forward observations by two periods.

**Seasonal variables** can be created with `gen seasonal_sales = S.sales`. This command will calculate the difference in `sales` between each observation and the previous observation. After running this command, we have:

| year | id | sales | lagged_sales | lagged2_sales | forward_sales | seasonal_sales |
|---|---|---|---|---|---|---|
| 2000 | 1 | 100 | . | . | 110 | . |
| 2001 | 1 | 110 | 100 | . | 120 | 10 |
| 2002 | 1 | 120 | 110 | 100 | 130 | 10 |
| 2003 | 1 | 130 | 120 | 110 | 140 | 10 |
| 2004 | 1 | 140 | 130 | 120 | . | 10 |

The first value in `seasonal_sales` is missing because it has no "previous" observation with which to take the difference. `S2.sales`, `S3.sales`, etc., will compute the difference in `sales` between observation $t$ and $t-2$, and between observation $t$ and $t-3$, etc.

**Difference variables** can be computed and are similar to seasonal variables. `D.sales` also compute "differences", and `D1.sales` and `S1.sales` are indeed equivalent. However, for periods greater than 1 (like `D2.sales`), the behavior is different.`D2.sales` would compute a "difference of differences", which means it will compute the following for observation number $t$:

$$(Sales_t - Sales_{t-1}) - (Sales_{t-1} - Sales_{t-2})$$

Again, after running `gen diff2_sales = D2.sales`, the first two observations of `diff2_sales` would contain missing values.

## 5.8 Display Expressions and Stored Results

The `display` command, followed by any arithmetic operation(s), allows you to use Stata like a calculator. This command simply displays whatever results follow. For example, we can apply `summarize` to the S&P 500 example dataset, and we can decide to calculate the mean of open divided by the mean of volume. We can just read off these values from the table and use `display 1194.884/12320.68` to obtain the result, but we can also directly get these values without reading off the table by using stored results.

Taking a look at the help file for `summarize` (`help summarize`) will lead us to discover that every time the `summarize` command is run, there are several results that are internally stored. This includes the number of observations (`r(N)`), the mean (`r(mean)`), and more. Thus, running `summarize change`, and then `display r(N)` will show us that there are 247 (non-missing) values in the `change` variable. The dataset in its entirety has 248 observations, but the first observation has a missing `change` value.

If we then run `summarize` for another variable, the stored results like `r(N)` and `r(mean)` will be overwritten with the results of the new variable.

`display` can also handle and output strings: `display "this is a string"`. Note that strings must be placed inside quotations for Stata to recognize them.

## 5.9 Date Variables

We have seen in the S&P 500 example dataset an example of the date variable. What allows a variable to be a date is a special format detectable in the Properties section of the Stata window. Most numerical variables have the format "%9.0g". We can discover from `help format` that dates take the format "%td". Thus, changing any numerical variable's format to "%td" will turn it into a date variable. This can be done via `format num_var %td`, where `num_var` is the name of a numerical variable.

Stata dates start from January 2, 1960, corresponding to a numerical value of `1`. A numerical value of `2` expressed in date format would be January 3, 1960, and so on.

The specific format of how dates are written can also be customized, by clicking the "..." button to the right of the date variable's "Format", in the Properties section.

## 5.10 Indicator and Interaction Variables

**Indicator variables** are variables that mark whether an observation satisfies some criterion, and these variables only take the values `0` or `1`. `1` would denote that the observation satisfies that variable's criterion, and `0` represents otherwise. Indicator variables can be created by going to *Data > Create or change data > Other variable-creation methods > Create indicator variables*. If we use the `citytemp.dta` example dataset, we can select `division` here as the variable to tabulate, and indicator variables will be generated for each division that exists. That is, the number of indicator variables created will be the same as the number of divisions there are. Each indicator variable (column) will denote if a particular observation *is* that division (with a `1`) or not (`0`). The "New variables' stub" simply indicates what word you would like to precede each of the new variables' names.

Now, let's reload the `citytemp.dta` dataset by running `sysuse citytemp.dta, clear`. Then, we can see the interaction between the variables `division` and `region` by using the `tabulate divison region` command. The output table will show how many observations are in each `division-region` combination. We wish to create **interaction variables** that represent this effect. Go to *Data > Create or change data > Other variable-creation commands > Interaction expansion*. Select the second option, and choose the two variables you would like to see interacted (`division` and `region`). Once again, Stata will create a lot of new variables, so the prefix for these variables can be specified in the last setting. As an example, we set this prefix to "`_int`". Hit OK and go to `browse`. What Stata has done is create dummy variables which are equivalent to the indicator variables for each `division`, and also for each `region`. Then, to create the interaction variables *between* `division` and `region`, Stata easily multiplies the relevant columns element-wise, to obtain `0`s and `1`s in the interaction variable column. For example, for the `int_divXreg_2_2` column, these variable values were obtained by multiplying the `int_divisio_2` column by the `int_region_2` column. A `1` would represent an observation that is in division 2 and region 2.

Note that the equivalent command for generating these interaction variables was `xi I.division*I.region, prefix(int_)`.

## 5.11 Filling in Missing Values

Remember that missing values are represented by a dot (`.`) in Stata. Suppose we are using the `sp500.dta` dataset, which includes the variables `date` and `open`. However, suppose that in our case, `open` has a few missing values. You can allow Stata to perform internal operations, using existing information from non-missing values to interpolate what those missing values *might* be. Those missing values will then be filled in. Go to *Data > Create or change data > Other variable-creation commands > Linearly interpolate/extrapolate values*.

## 5.12 Frequency Weights

Frequency weights refer to the count of how many times a certain value is represented. Consider the following dataset:

| sales | fw |
|---|---|
| 100 | 5 |
| 150 | 1 |
| 200 | 1 |

The mean of the `sales` column is clearly `150` (`mean sales`), but if we wish to consider a weighted average of the `sales` column, with the weights given by the `fw` column, we can type in `mean sales [fweight = fw]`. That is, the mean displayed by this command will be calculated by $\frac{5(100)+1(150)+1(200)}{5+1+1} = 121.43$. Also as expected, the number of observations is indicated to be 7, rather than 3.

## 5.13 Sampling Weights

A column of sampling weights, or probability weights, indicate the inverse of the probability that each observation can be sampled from the population. Suppose we have the dataset:

| sales | pw |
|---|---|
| 100 | 0.9 |
| 150 | 0.1 |
| 200 | 0 |

According to this dataset, there is a 90% probability of sampling the $100 sales observation, a 10% probability of sampling the $150 sales observation, and a zero probability of sampling the $200 sales observation. The mean, as calculated under the sampling weights, is given by `mean sales [pweight = pw]`.

Check the `help` files to see which commands allow the usage of certain weights. For example, `help summarize` will tell you that the `summarize` command does not allow the usage of `pweight`, but does allow the usage of `fweight`.

## 5.14 by

Remember that the `if` statement can be used after commands to apply a condition. The command will only be run on observations that meet the condition specified by the `if` statement. For example, consider the system `citytemp.dta` dataset. `summarize tempjan` will give summary statistics for all observations in the `tempjan` variable, but `summarize tempjan if region == 1` will give `tempjan`'s summary statistics for only the observations where the region is "NE", which is represented by `1`.

On the other hand, `by` is a prefix that can be used to replicate a command for each category in the `by` variable. For example, if you type `by region: summarize tempjan`, the `summarize tempjan` command will be applied to only observations where `region == 1`, followed by where `region == 2`, where `region == 3`, and where `region == 4`. You will see that four sets of summaries are displayed, one for each region.

`bysort` is simply the `by` command with the `sort` option, which sorts values in ascending order.

## 5.15 String Conversion

In the `browse` window of any dataset, black-colored characters denote numerical values, and red-colored characters denote string values. However, even if a certain variable has only numbers visible, but `browse` displays them in red color, the variable is stored in a string format. To convert such a string variable `str`, which is actually filled with numbers, to a numeric, go to *Data > Create or change data > Other variable-transformation commands > Convert variables from string to numeric* and select the desired options. Choosing `str` in the "variables to convert" field and creating a new variable is equivalent to the command `destring str, generate(new_var_name)`. This will not replace the original `str` variable, but rather generate a new

variable `new_var_name` that is the numeric version of the original. If you want to replace the original `str` variable, use `destring str, replace`.

Conversely, `tostring numeric, generate(new_var_name)` will convert `numeric` to a newly generated string variable.

## 5.16 `real()`

A drawback to the `destring` command is that if the targeted string variable contains even a single non-numeric character inside any observation of that variable, conversion will not work. No new variable can be generated, because Stata cannot convert a letter, for example, to a proper number.

However, `real()` is a function that is more versatile in this respect. We will learn more about functions later on, but for now, just know that what goes inside the parentheses `()` are function **arguments**. Functions take arguments as input, performs some action, then returns the result as output. In other words, functions are *applied* to its arguments.

The `real()` function will transform values in its argument to real numbers and return them as numerics. To generate a new variable by converting the variable `str` to numerics, use `gen new_var_name = real(str)`. Then, for any observations in which there were one or more characters in `str`, missing values will appear in `new_var_name`. However, for all the rest of the observations in which there were only numbers in `str`, the appropriate numeric values will be present in `new_var_name`.

# 6 Milestone Project: Hospitals Data

The purpose of this milestone project is to consolidate all our previous knowledge, and put it to use on a new dataset to analyze. We will be using the `k1.dta` dataset, available in the course materials. First, open the dataset from the *File* menu.

We seek an overview of the dataset's variables by entering the command `describe`. We see that the dataset contains information on patients' treatment data, including their hospital information, their diagnoses, their treatments, etc. We also see that there are 26131 observations, so to save memory, we enter the command `compress` in order to allow Stata to optimize storage space.

We can also `browse` the dataset in a table format, and we see that for one hospital, there are many patients. It is thus useful to move the `Hospital_ID` variable to the front, but we can also choose to move `Patient_ID` and `Case_ID` to the front, using `order Hospital_ID Patient_ID Case_ID`. Then, we can order the dataset by `Hospital_ID`, `Patient_ID`, and `Case_ID` in that priority order: `sort Hospital_ID Patient_ID Case_ID`.

## 6.1 Hospitals

**Question**: How many unique hospitals are represented in the dataset?

```
codebook Hospital_ID, compact
```

This will show that there are 164 unique hospitals. As shown by the help file for `codebook`, the `compact` option displays a compact report.

**Question**: How many observations are there for each hospital?

```
tab Hospital_ID, sort
```

From this output, we will see that hospital #79 has the most number of observations (1353).

Let's take a look at the observations for hospital #79:

```
list if Hospital_ID == 79
```

## 6.2 Patients

We can see the frequency of patients from hospital #79 by:

```
tab Patient_ID if Hospital_ID == 79, sort
```

A few patients entered hospital #79 three times, some entered twice, and most entered once.

## 6.3 Cases

Similarly, we can see the frequency of cases from hospital #79:

```
tab Case_ID if Hospital_ID == 79, sort
```

We have just one line for each case. So, each time a patient entered the hospital, they presented a different case.

```
codebook Patient_ID Case_ID if Hospital_ID == 79
```

There were 1276 unique patients and 1353 unique cases. The excess number of cases over the number of patients is as expected, since some patients had more than one case.

## 6.4 Years

```
codebook year
```

From this, we can see that the dataset represents two years of information.

## 6.5 Duration of Stay

```
summ Duration_Stay
```

The average duration of stay was 9.938732 days, with a standard deviation of 11.7.

On the other hand, for only hospital #79:

```
summ Duration_Stay if Hospital_ID == 79
```

The average duration of stay for patients in hospital #79 was 6.24612 days, with a standard deviation of 10.7. Both statistics are lower than the overall dataset's.

## 6.6 Sex

```
summ sex if Hospital_ID == 79
```

Note that the `sex` variable is stored as `0`s and `1`s, with `0`s representing females and `1`s representing males. Thus, the mean displayed by the summary output for this line of code is, in fact, the proportion of males in hospital #79's patients. The mean is 0.66, so more males than females visited hospital #79.

## 6.7 `Duration_Stay` vs. `Emergency_Stay`

Let's reorder the dataset, bringing to the front the following variables:

```
order year Hospital_ID Patient_ID Case_ID Duration_Stay sex age Nationality Emergency_Stay
> race Mortality black
```

We can summarize the `Duration_Stay` and `Emergency_Stay` variables for `race == 1` (Swiss) by:

```
summ Duration_Stay Emergency_Stay if race == 1
```

This summary is for all hospitals. If we want to add the condition that only observations for hospital #79 be summarized, we use the "and" (`&`) operator:

```
summ Duration_Stay Emergency_Stay if race == 1 & Hospital_ID == 79
```

Suppose we want to calculate the variation coefficient for `Duration_Stay` and `Emergency_Stay`, which is given by: Variation Coefficient $= \frac{\text{Standard Deviation}}{\text{Mean}}$. We want to extract the mean and standard deviation for `Duration_Stay` and `Emergency_Stay` from the previous summary table, and we can do so using the stored results.

According to the `help` file for `summ`, we can retrieve the mean by using `r(mean)` and the standard deviation by using `r(sd)`. The values stored in this code will be statistics for the *last* variable involved in the calculation. If we type `display r(mean)`, the number displayed will represent the average `Emergency_Stay` value for observations where `race` is Swiss and `Hospital_ID` is 79. Thus, to calculate the variation coefficient for `Duration_Stay`, we have to remove `Emergency_Stay` from the code:

```
summ Duration_Stay if race == 1 & Hospital_ID == 79
```

Then, the variation coefficient for `Duration_Stay` (for Swiss patients from hospital #79) can be given by:

```
display r(sd) / r(mean)
```

The result is 1.7473046.

We repeat the process for `Emergency_Stay`:

```
summ Emergency_Stay if race == 1 & Hospital_ID == 79
display r(sd) / r(mean)
```

The result is 3.3641698, representing more dispersion than `Duration_Stay`.

## 6.8 Mortality Rates

First, we summarize `Mortality`:

```
summ Mortality
```

Comparing the average population mortality rate with the mortality with the largest hospital, hospital #79,

```
summ Mortality if Hospital_ID == 79
```

We see that the largest hospital has a much lower mortality rate than the overall.

```
summ Mortality if Hospital_ID == 79 & black == 1
```

From this output, we see that the mean mortality rates for blacks and the population are very similar, indicating likely no effect of being black on the mortality rate.

```
summ Mortality if Hospital_ID == 79 & black == 1 & year == 2010
```

This mean is also very similar to the above.

## 6.9 Changing Value Labels

As an example, the value label for 0 in the variable `race` is currently "Non-swiss". Suppose we want to change this to "Non-Swiss". Go to *Data > Data utilities > Label utilities > Manage value labels*, and select from the drop down menu the label you'd like to modify. Double click that label, and proceed to change it. `codebook race` will now show the corrected label.

## 6.10 Age Effects

We want to study the effects of `age` on mortality rates. `tab age` will show that there are 7 age categories in the dataset. Again, `age` is stored as integers but has value labels. If we want to look at mortality rates for all age groups in hospital #79:

```
summ Mortality if Hospital_ID == 79
```

If we want to look at mortality rates for the oldest age group (`age == 7`) in hospital #79, we should use:

```
summ Mortality if Hospital_ID == 79 & age == 7
```

We see that there is only one patient in this summary.

If we want to look at mortality rates for the second-oldest age group (`age == 6`) in hospital #79, we should use:

```
summ Mortality if Hospital_ID == 79 & age == 6
```

We see that the average mortality rates for this age group is higher than the average mortality for all ages.

For the youngest age group:

```
summ Mortality if Hospital_ID == 79 & age == 1
```

As expected, for the youngest patients, the mortality rate is very low.

For black patients in the youngest age group:

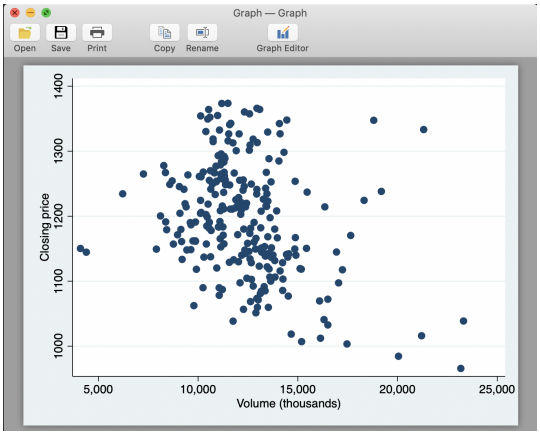
```
summ Mortality if Hospital_ID == 79 & age == 1 & black == 1
```

We see that race has very little effect on mortality rates for the youngest age group.

# 7 Graphs

## 7.1 Scatter Plot

Let's use the S&P 500 sample dataset. To create a scatter plot, go to *Graphics > Twoway graph (scatter, line, etc.)*, click "Create…", and select "Scatter" under basic plots. Scatter plots are useful to inspect the relationship between two variables and check for any linear associations, positive or negative. Suppose we want to inspect the relationship between the `close` and `volume` variables. Select `close` for the y-variable and `volume` for the x-variable, and click "Accept".




Press "OK" to see the graph! The points appear randomly scattered, so there does not seem to be any linear relationship between `close` and `volume`.

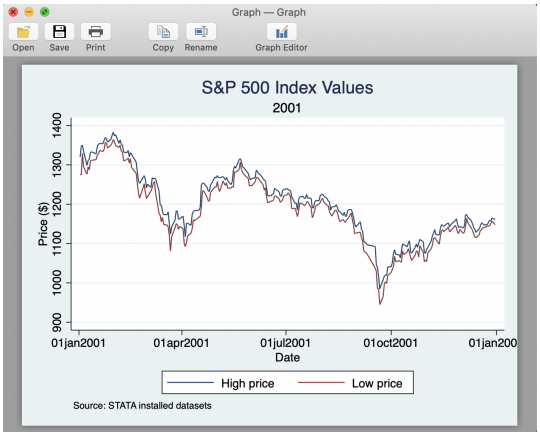
If we close the graph window and go to *Graphics > Twoway graph (scatter, line, etc.)* once again, we'll see that "Plot 1" is still available. Stata has stored this graph in memory for you to access repeatedly. Clicking "Edit" will allow you to change the type and variables of the graph, and the menu in the top bar of this window allows you to change properties of the graph, such as the title, subtitle, axis labels, and much more. Note that by default, the axis labels will use the stored variable labels, and so will the legend, if you choose to select one in the "Legend" tab.

The resulting graph can easily be copied and pasted to other applications by right-clicking on the graph itself. Also note that for every menu command used, Stata displays the corresponding code version in the "Results" section of the main window, which can be copied to a do-file to replicate execution.

## 7.2 Line Graph

Line graphs are useful for representing trends over time. We can turn "Plot 1", created in the previous section, into a line graph, by going to *Graphics > Twoway graph (scatter, line, etc.)*, selecting "Plot 1", and clicking "Edit". Choose "Line" under basic plots, and let's observe the trend of the `high` variable over time by selecting `high` as the y-variable and `date` as the x-variable.

If we want a second line on the same graph, simply go to the twoway graph menu and create a new plot. Again, select "Line", but this time put `low` as the y-variable (and `date` as the x-variable). Now, we have "Plot 1" and "Plot 2". Clicking OK will show both lines on the same graph, with an automatically generated legend.




## 7.3 Combining Graphs

Suppose that using twoway graph's menu bar, we also added a y-axis label, x-axis label, title, subtitle, and note to the graph:

- **Y-Axis Label**: `Price ($)`
- **X-Axis Label**: `Date`
- **Title**: `S&P 500 Index Values`
- **Subtitle**: `2001`
- **Note**: `Source: STATA installed datasets`

The equivalent code version for creating the line graph with two lines and the above properties are as follows:

```
twoway (line high date) (line low date),
> ytitle(Price ($)) xtitle(Date) title(S&P 500 Index Values)
> subtitle(2001) note(Source: STATA installed datasets)
```

Note that if you to place this in a do-file, the three lines above must be all in one line to be executed properly. Here, a line break necesssary for formatting, but which must not be a line break in a do-file, is denoted by a "`>`" sign.

To save this graph in memory, run the following command immediately after creating the graph:

```
graph save g1.gph, replace
```

This saves the aforementioned line graph under the filename `g1.gph`, while replacing another graph potentially with the same name in Stata's memory.

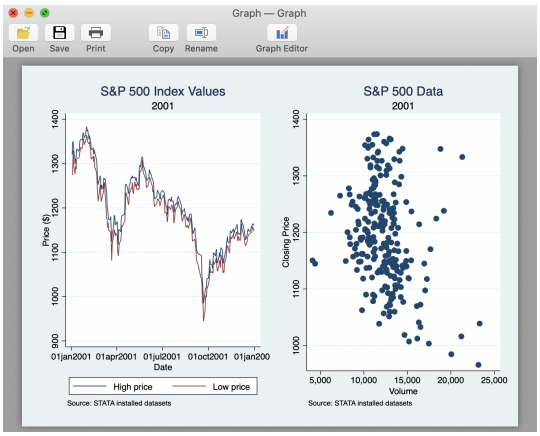
Following the format of Stata's code for a line graph, the code for a scatter plot of the `close` and `volume` variables are as follows.

```
twoway (scatter close volume),
> ytitle(Closing Price) xtitle(Volume) title(S&P 500 Data)
> subtitle(2001) note(Source: STATA installed datasets)
graph save g2.gph, replace
```

To combine `g1.gph` and `g2.gph` into one graph, simply run the command:

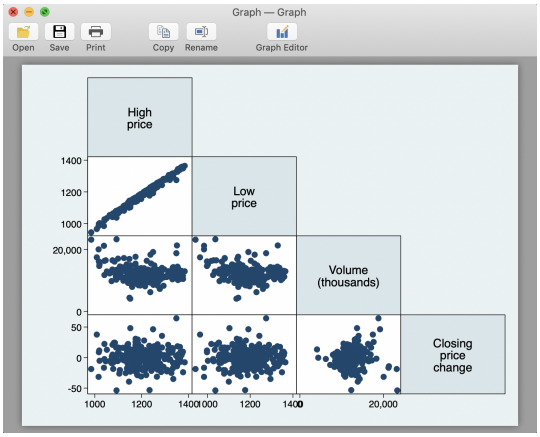
```
graph combine g1.gph g2.gph
```

The two graphs will appear side-by-side.




## 7.4 Scatterplot Matrix

A scatterplot matrix shows the linear relationships between all combinations of pairs of variables. To create a scatterplot matrix, go to **Graphics > Scatterplot matrix**, and enter all the variables for which you'd like to see relationships. For example, enter `high`, `low`, `volume`, and `change`, select the option "Lower triangular half only", and press "OK".



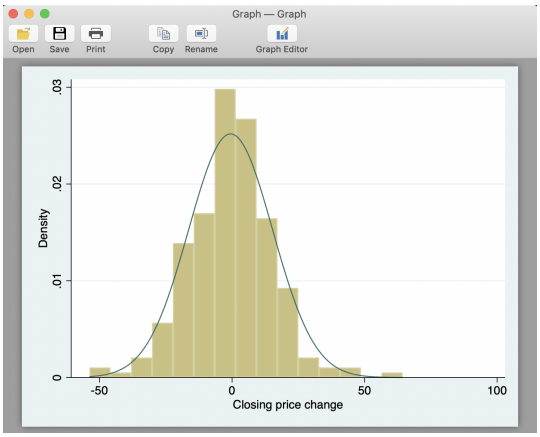

From the resulting output, you can see that each variable is represented in the scatterplot matrix. For example, the plot right beneath the "High price" label represents the scatterplot with `high` on the x-axis and `low` on the y-axis. There is a very strong, positive relationship here.

The second plot beneath the "High price" label represents the scatterplot with `high` on the x-axis and `volume` on the y-axis. The plot to the right of that is the scatterplot with `low` on the x-axis and `volume` on the y-axis, and so on.

For any single scatterplot, the label matching with the horizontal edge of that block represents the x-axis label, and the label matching with the vertical edge of that block represents the y-axis label.

## 7.5 Histogram

A histogram draws closed bars that represent the frequency, or density, or a category appearing in a variable. To draw a histogram in Stata, go to *Graphics > Histogram*, and select the `change` variable as an example. Pressing "OK" will show the histogram, which appears to be normal. We can further verify normality by adding a normal density curve, by going to *Graphics > Histogram*, selecting "Density plots" in the menu, and selecting "Add normal-density plot".




It appears that the closing price change is quite normal. We could also have chosen to add a kernel density line, which is the fitted density.

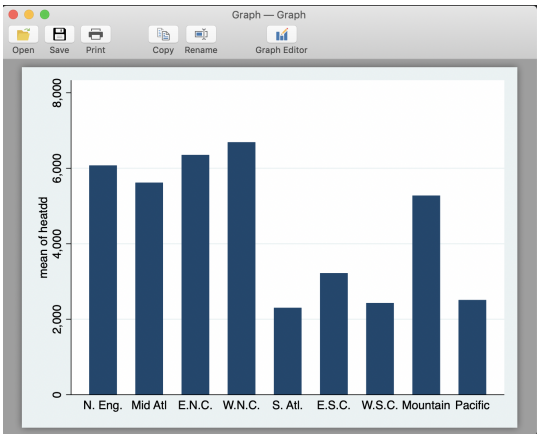
From the equivalent code, we can see that Stata defaults to 15 bins for the histogram, but we can change this behavior. In the histogram window, you can change the number of bins to 20, and the appearance is less normal.

Also note that by default, the histogram has density as its y-axis, which means that the sum of areas of all bars will be equal to 1. We can instead to plot frequencies, by selecting the "Frequency" option under "Y axis" in the histogram window.

## 7.6 Bar Graph

Let's use the `citytemp.dta` example dataset.

To create a bar graph, go to *Graphics* > *Bar chart*, and select the variable you'd like to plot (`heatdd` in our example). Since each bar represents a category, we must select a grouping variable under the "Categories" tab in the menu bar. Select `division` as the grouping variable, and plot the mean for `heatdd` within each `division`.




As usual, we can put additional information in the title, axis labels, etc. We can also opt for a horizontal layout of the bars in the "Main" tab.

## 7.7 Schemes and Colors

Let's use the `sp500.dta` example dataset and create a line graph of `close` against `date`.

```
sysuse sp500.dta
twoway (line close date)
```

We can change the scheme of this graph by going to *Graphics > Change scheme/size*, and selecting the desired scheme. The scheme controls the appearance of the overall graph, including line colors, as you will discover when experimenting.

Next, let's create a line graph of the variables `open`, `high`, `low`, and `close`, against `date`. A shortcut command is:

```
line open high low close date
```

Changing schemes here will also include changing the line types for each variable.

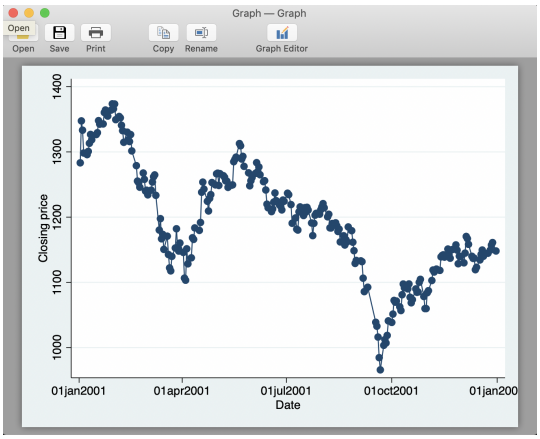
Margins may also be customized from this menu, since axis labels are currently cut off by the window. For example, the large margin works well in this case. You can also adjust the height and width of the graph.

Note that schemes for twoway graphs can also be found under the "Overall" tab in *Graphics > Twoway graph (scatter, line, etc.)*. The "Scale text, markers, and lines" option is available for you to change the size of graph elements. The scale is a multiplier that indicates how large elements should be. For example, a 0.8 entry here will change all graph elements to 80% of their default size (smaller).

## 7.8 Connected Graphs

From *Graphics > Twoway graph (scatter, line, etc.)*, we can create a new plot which is a connected graph. Select "Connected" under basic plots, with `close` as the y-variable and `date` as the x-variable.

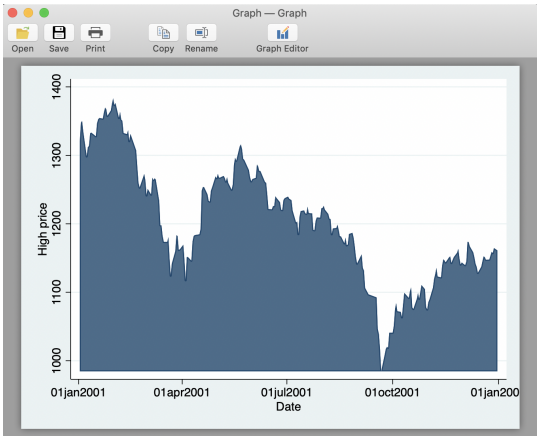
```
twoway (connected close date)
```




## 7.9 Area Graphs

We select the "Area" basic plot in *Graphics > Twoway graph (scatter, line, etc.)*, and `high` for the y-variable and `date` for the x-variable.

```
twoway (area high date)
```

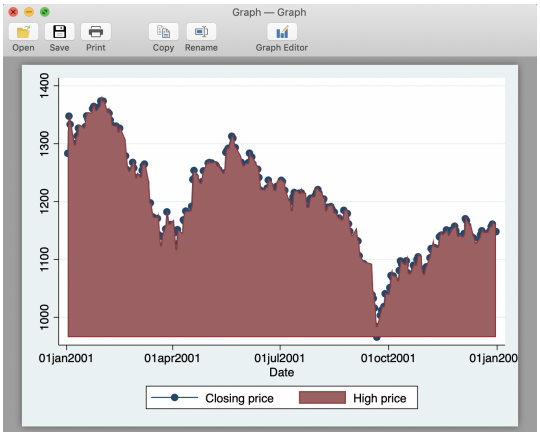



## 7.10 Overlay Graphs

Sometimes, we wish to overlap a certain plot type on top of another plot, and this can easily be done so by simply creating more than one graph in *Graphics > Twoway graph (scatter, line, etc.)*. Pressing OK will display all the plots in a single graph. However, since you will have multiple variables represented on the graph, most likely the y-axis label will be dropped, in which case you would want to opt for a legend, as well as a custom y-label. The "Move Up" and "Move Down" options allow you to reorder the stack of plots when displayed, with the bottom-most graph appearing on top when displayed. You can think of this ordering from top to bottom as the order in which plots are "drawn" into the window. The top-most plot is drawn first, so it will appear behind all the rest of the plots in the window. The bottom-most plot is drawn last, so it will appear in front all the rest of the plots. This is apparent in the code version:

```
twoway (connected close date) (area high date)
```

Here, the area graph will be drawn after (on top of) the connected graph.




For a certain variable's plot whose scale may not fit well with the other variables, you can add a second y-axis on the right of the graph by selecting this option in "Create..." of the *Graphics > Twoway graph (scatter, line, etc.)* window.

# 8 Descriptive Statistics

## 8.1 Mean

To calculate the mean for a certain variable, you can either use the `summarize` command on that variable, or the `mean` command:

```
mean var
```

Equivalently, you can go to *Statistics > Summaries, tables, and tests > Summary and descriptive statistics > Means*, and select the desired variable. The advantage of using the `mean` command is that Stata will automatically calculate the standard error for that estimate (sample mean), as well as the 95% confidence interval. The 95% confidence interval indicates the interval within which we are 95% confident that the true mean will lie.

`summarize` and `mean` calculate the arithmetic means, but Stata can also calculate geometric and harmonic means. Go to *Statistics > Summaries, tables, and tests > Summary and descriptive statistics > Arith./geometric/harmonic means*, select the desired variable, and click OK. The equivalent command is `ameans`.

## 8.2 Tabulating Data

### 8.2.1 One-Way Table

Tabulating data means to display frequency tables of how many observations fall within a certain category of a variable, like `division` for the `citytemp.dta` example dataset. To tabulate data, go to *Statistics > Summaries, tables, and tests > Frequency tables > One-way table*, and select the categorical variable for which you'd like to see frequencies. The equivalent command is:

```
tabulate division
```

Often, a useful option is to check "Display the table in descending order of frequency" in the one-way table menu, which shows the category of the highest frequency at the top of the table. The equivalent code is:

```
tabulate division, sort
```

As categorical variables are stored as integers with labels, you can also choose to omit the labels and instead display the integers by checking "Display numeric codes rather than value labels", or equivalently:

```
tabulate division, nolabel sort
```

It really only makes sense to tabulate categorical variables, not numerical variables. If you have a variable containing data on temperatures, tabulating this variable will display the number of observations that match a certain temperature, which is not very useful.

### 8.2.2 Two-Way Table

Using the `citytemp.dta` example dataset, suppose we want to see the number of observations in each *combination* of divisions and regions, we can use a two-way table. A two-way table will list the categories of each of the two variables are rows and columns, and the contents of the table will represent the number of observations that fall in each pairing. To create a two-way table, go to *Statistics > Summaries, tables, and tests > Frequency tables > Two-way table with measures of association*, and select the row and column variables.

The equivalent command is simply the keyword `tabulate`, followed by two variables.

```
tabulate division region
```

### 8.2.3 Table of Summary Statistics

Aside from `summarize`, you can customize the summary statistics for whichever variables you'd like described in a table. Go to *Statistics > Summaries, tables, and tests > Other tables > Compact table of summary statistics*, and select the desired variables and summary statistics you would like displayed, and press OK.

For example, if we want to calculate the mean, count, range, and coefficient of variation of the `heatdd` variable for *each* division, we can select `heatdd` as the variable, check "group statistics by" `division`, and choose the four summary statistics in the "Statistics to display" section. The equivalent command would be:

```
tabstat heatdd, statistics( mean count range cv ) by(division)
```

As the name suggests, the other option, *Statistics > Summaries, tables, and tests > Other tables > Flexible table of summary statistics*, gives even more flexibility in defining summary statistics to display. You can experiment with this option on your own.

## 8.3 Correlation and Covariance Matrices

Let's use the `sp500.dta` example dataset.

To quickly display correlation or covariance matrices in Stata, go to *Statistics > Summaries, tables, and tests > Summary and descriptive statistics > Correlations and covariances*, and select the variables for which you'd like to see correlations. For example, selecting `open`, `high`, and `low` will show very high correlations between each pairing combination of the three variables, since they are very similar ideas of the S&P 500 index. The equivalent command is:

```
correlate open high low
```

To obtain the covariance matrix, simply go to the "Options" tab of this menu and opt to display covariances. Or, the code version is with the `covariance` option:

```
correlate open high low, covariance
```

Note that if there are any values missing for a certain variable in an observation, Stata will ignore that observation entirely when calculating correlations or covariances involving that variable. You may test this by replacing some of the data in the `sp500.dta` example dataset with the code `replace varname = . in 50/100`, as an example (this will replace the 50th to 100th values in the variable `varname` with missing values).

## 8.4 Confidence Intervals

Stata can compute confidence intervals for various descriptive statistics. Go to *Statistics > Summaries, tables, and tests > Summary and descriptive statistics > Confidence intervals*, and select the desired statistic, corresponding variable(s), and the desired confidence level. You can also choose to assume that the statistic is normally distributed by the Central Limit Theorem and large sample approximation, or Poisson-distributed, etc.

*Note*: The Central Limit Theorem states that given a large enough sample, the mean of the sample is a random variable that is normally distributed.

For example, with the `sp500.dta` example dataset, if we wish to compute the 95% confidence interval for the mean of the `close` variable, the equivalent code is:

```
ci means close
```

The 95% confidence interval for the variance of the `change` variable is given by:

```
ci variance change
```

If you ever have any doubts about how Stata is computing confidence intervals, such as by assuming the normal distribution, check the help file for the command `ci`. In the case of normally distributed means, Stata computes the confidence interval by the sample mean, plus or minus the standard error times the t-quantile.

The reason that the t-quantile is used here instead of the standard normal quantile is that variance of the data is unknown, and must be estimated. To account for this fact, the Student's t distribution is used instead of the standard normal distribution.

Aside from calculation of the confidence interval based on some variable available in the dataset, you can also compute theoretical confidence intervals for some arbitrary, not readily available data. In *Statistics > Summaries, tables, and tests > Summary and descriptive statistics > Normal mean CI calculator*, you can provide Stata with customized information about your imaginary sample size, sample mean, and sample standard deviation, and Stata will act as a calculator to compute your desired level of confidence interval, for whichever type of statistic you select.

For example, if we wish to compute the 97.5% confidence interval for an imaginary sample's mean, and such a sample has size 100, mean 10, and standard deviation 8, we would select in this window "Normal mean" and set the appropriate options. Pressing "OK" will let the calculator compute the confidence interval. The equivalent code is:

```
cii means 100 10 8, level(97.5)
```

As a side note, higher confidence levels will generate wider confidence intervals, i.e. confidence intervals of a wider range. This is quite intuitive, as a 95% confidence interval would denote the range for which we have a 95% confidence in which the true statistic falls. If we want a higher confidence of the true statistic falling in the interval, we must have a wider interval, since a wider interval is more likely to encompass the true value.

# 9 Functions

## 9.1 Math Functions

Functions are slightly different from the commands that we've been using up to this point. There are many types of functions in Stata, divided for organization purposes. You can view all the types of functions by typing `help functions`. In this section, we will go over the "mathematical functions".

Math functions typically take numeric data types as argument.

### 9.1.1 `abs(x)`

Functions take in arguments, and for the `abs` function, the absolute value of the numeric argument `x` will be computed.

```
display abs(10)
```

This will display 10.

```
display abs(10)
```

This will also display 10.

Note that the `display` command is required here in order to *output* the return value of the function. The sole portion `abs(10)` will *return* the value `10`, but in order for this return value to be outputted, we must pair the return value with the `display` command. This also provides insight into the difference between a command and a function.

The `abs()` function can also take a variable (of a dataset) as argument. Suppose we have the following dataset:

```
var1
1
-2
3
5
-9
-8
```

The following command will generate a new variable containing the absolute values of `var1`.

```
gen var_abs = abs(var1)
```

The dataset is now:

```
var1    var_abs
1       1
-2      2
3       3
5       5
-9      9
-8      8
```

### 9.1.2 `ceil(x)`

The ceiling function rounds the argument `x` up.

```
display ceil(10.4)
display ceil(10.000001)
display ceil(11)
```

The output of all three lines is 11.

Suppose we have the dataset:

```
var2
10
2.2
3.6
5.56
```

We run the `ceil()` function on `var2`, generating a new variable.

```
gen var_ceil = ceil(var2)
```

The dataset is now:

```
var2     var_ceil
10       10
2.2      3
3.6      4
5.56     6
-4.2     -4
```

#### 9.1.3 `floor(x)`

The `floor()` function is the counterpart of the `ceil()` function, rounding down instead.

#### 9.1.4 `comb(n,k)`

The `comb()` function returns the number of ways of choosing k elements from n elements. The corresponding mathematical operator is the "choose" function.

```
display comb(10,2)
```

This will output the result of 10 choose 2, which is 45.

#### 9.1.5 `exp(x)`

The exponential function of `x` calculates $e^x$.

#### 9.1.6 `ln(x)`

This is the inverse of the exponential function, calculating $\ln(x)$. `log(x)` is equivalent to `ln(x)`.

#### 9.1.7 `int(x)`

The `int()` function rounds the argument `x` towards 0. For example:

```
display int(3.6)
```

The output is 3.

```
display int(2.2)
```

The output is 2.

```
display int(-4.2)
```

The output is -4.

#### 9.1.8 `sum(x)`

The `sum(x)` function applied to a variable computes the *cumulative* sum, going down the observations. Take the following dataset as an example:

```
var3
1
2
3
4
5
```

Let's generate a new variable with the `sum()` function.

```
var3    var_sum
1       1
2       3
3       6
4       10
-5      5
```

So, each value in `var_sum` is computed by summing all the previous values (inclusive) in the argument variable `var3`.

#### 9.1.9 Other Functions

For more, consult the help file for mathematical functions, directly accessible via `help math_functions`. Note that some functions have special requirements for the argument, such as the `logit(x)` function requiring `x` to be in the interval from 0 to 1.

### 9.2 String Functions

Stata has many useful functions for string manipulation. Remember that these functions are applicable to string variables, not just single string values.

#### 9.2.1 `abbrev(s,n)`

The `abbrev()` function takes the string `s` and abbreviates it to `n` characters. The method that Stata follows is for strings longer than `n` characters:

- Keep the first $n-2$ characters of the string `s`
- Add a tilde (~)
- Use the last character of the string `s`.

For example, according to the help file:

```
display abbrev("displacement", 8)
```

The output is `displa~t`.

#### 9.2.2 `char(n)`

Each ASCII character has a corresponding numerical representation, and `char(n)` will display the corresponding character for the number `n`.

#### 9.2.3 `uchar(n)`

As opposed to `char(n)`, `uchar(n)` will return the Unicode character for the number `n`.

#### 9.2.4 `plural(n,s)`

This function pluralizes the string `s` based on the count in `n`. That is, if `n` is greater than or equal to 2, the function will return `s` followed by an additional character "s".

#### 9.2.5 String Concatenation

To concatenate two strings, simply use the "+" operator:

```
display ("a" + " " + "b")
```

The output is `a b`. Note that here, the string addition must be within parentheses for Stata to recognize the operation as a function.

#### 9.2.6 String Multiplication

Multiplying a string by $n$ will repeat the string $n$ times. For example:

```
display ("hello" * 3)
```

The output is `hellohellohello`.

#### 9.2.7 `stritrim(s)`

If a string `s` has any consecutive spaces (blank characters), this function will remove them.

```
display stritrim("Is this     okay?")
```

The output is `Is this okay?`.

#### 9.2.8 `strlen(s)`

This function counts the number of characters in `s`.

#### 9.2.9 `strlower(s)`

This function returns the string `s` in all lower case letters. If you have Unicode characters in the string, you can use `ustrlower(s)`.

```
display ustrlower("ENCHANTÉ")
```

The output is `enchanté`.

If you used the non-Unicode version of the function for a string that contains Unicode characters, then only the non-Unicode characters will be taken care of.

```
display strlower("ENCHANTÉ")
```

The output is `enchantÉ`.

#### 9.2.10 `strupper(s)`

This function returns the string `s` in all upper case letters. If you have Unicode characters in the string, you can use `ustrupper(s)`.

#### 9.2.11 `strltrim(s)`

This function removes all leading whitespace characters in the string `s`. If you have Unicode characters in the string, you can use `ustrltrim(s)`. "`strltrim`" stands for "string left trim".

```
display strltrim("  Hi")
```

The output is `Hi`.

#### 9.2.12 strrtrim(s)

This function removes all trailing whitespace characters in the string `s`. The Unicode version is `ustrrtrim(s)`. "`strrtrim`" stands for "string right trim".

```
display strltrim("Hi ")
```

The output is `Hi`.

#### 9.2.13 strtrim(s)

This function removes all leading and trailing whitespace characters in the string `s`. The Unicode version is `ustrrtrim(s)`.

```
display strltrim(" Hi ")
```

The output is `Hi`.

#### 9.2.14 strofreal(n)

This function converts a numeric argument `n` into a string. You can also specify the format of the resulting string; feel free check the help file.

#### 9.2.15 strpos(s1,s2)

This function locates the substring `s2` within `s1`, and returns the position of the first character satisfying the match. If no match is found, 0 is returned.

```
display strpos("Peter", "t")
```

The output is 3, since `t` is the third character in the string `Peter`.

#### 9.2.16 strrpos(s1,s2)

This function locates the *last* match of substring `s2` within `s1`, and returns the character position of the match.

```
display strrpos("Peter", "e")
```

The output is 4.

#### 9.2.17 ustrrpos(s1,s2[,n])

This is the Unicode version of `strrpos()`, but allows for an option argument `n`. According to the help file, "If `n` is specified and is greater than 0, only the part between the first Unicode character and the $n$-th Unicode character of `s1` is searched."

#### 9.2.18 strproper(s)

This function turns the string `s` into a proper noun, capitalizing the first letter in each world in `s`.

#### 9.2.19 strreverse(s)

This function reverses the order of characters in the string `s`. The Unicode version is `ustrreverse(s)`.

#### 9.2.20 strtoname(s[,p])

In Stata 13, a convention was introduced that disallows variable names to include spaces. So, this function fills in the spaces with underscores. If the optional argument `p` is specified, then as long as `p` is not 0, then the function also inserts a leading underscore into the string `s`. The Unicode version is `ustrtoname(s[,p])`.

#### 9.2.21 subinstr(s1,s2,s3,n)

This function returns s1, where the first n occurrences of s2 in s1 have been replaced with s3. An example directly from the help file:

```
display subinstr("this is the hour","is","X",2)
```

The output is thX X the hour.

#### 9.2.22 subinword(s1,s2,s3,n)

Very similar to subinstr(), this function only replaces s2 with s3 if s2 is an actual word, bound by whitespace.

```
display subinword("this is the hour","is","X",1)
```

The output is this X the hour.

#### 9.2.23 substr(s,n1,n2)

This function extracts the substring of s, starting at position n1 and lasting a length of n2. A negative number for n1 indicates the n1-th position counting from the end of the string, and a dot (.) for n2 extracts the entirety of the remaining string after the n1-th position. Examples from the help file:

```
di substr("abcdef",2,3)
```

The output is bcd.

```
di substr("abcdef",-3,2)
```

The output is de.

```
di substr("abcdef",2,.)
```

The output is bcdef.

```
di substr("abcdef",-3,.)
```

The output is def.

```
di substr("abcdef",2,0)
```

The output is the empty string "".

```
di substr("abcdef",15,2)
```

The output is the empty string "".

The Unicode version of this function is usubstr(s,n1,n2).

#### 9.2.24 word(s,n)

This function returns the n-th word in the string s. The Unicode version is also available.

#### 9.2.25 wordcount(s)

This function returns the number of words in the string s. The Unicode version is also available.

There are many, many more string functions you can learn more about from consulting the help documentation.

## 9.3 Random Number Functions

Next, we will learn how to draw random numbers in Stata. The help file for this topic can be accessed by entering `help functions`, and selecting "random-number functions". Since we're drawing random numbers, there's no need to load any datasets, but we'll need to set the number of observations.

```
set obs 1000
```

### 9.3.1 `runiform()` and `runiform(a,b)`

The `runiform()` function has two versions, one with arguments and one without. For the form of the function that does not take any arguments, we still have to include the parentheses when using the function. This is the basic difference between a command and a function.

The `runiform()` function with no arguments generates random numbers between 0 and 1, which are uniformly distributed. Note that "uniformly distributed" simply means that any number between 0 and 1 is equally likely to be drawn. We can use this function to create a variable of random numbers.

```
gen random = runiform()
```

If you choose to use the function with arguments, you must supply both arguments, `a` and `b`. This form of the function will generate random numbers between `a` and `b`, which are uniformly distributed.

### 9.3.2 `runiformint(a,b)`

This function draws only integers in the range from `a` to `b`, inclusive.

### 9.3.3 `rnormal()`, `rnormal(m)`, and `rnormal(m,s)`

These functions draw values from a normally distributed random variable. If you do not supply the argument `m` nor `s`, `rnormal()` by default draws from the standard normal distribution (mean 0, standard deviation 1). If you only supply `m`, the standard deviation is assumed to be 1. Otherwise, you can specify both the mean `m` and the standard deviation `s` of the normal distribution.

A normal distribution is not the same thing as a uniform distribution. Put simply, the mean of the normal distribution is the value that is most likely to be drawn, and values less than or greater than the mean are less and less likely to be drawn the further away they are from the mean.

The standard deviation, on the other hand, indicates how widely distributed the normal random variable is. A higher standard deviation would mean that values further away from the mean are more likely to be drawn.

## 9.4 Extended Variable Functions

Extended variable functions' documentation can be accessed via `help egen`. These functions are all useable with the `egen` command to generate new variables, which is an extension to the very commonly used `generate` command.

### 9.4.1 `total(exp)`

This function takes a numeric variable of a dataset as input and calculates the sum of all values in that variable. Technically, `exp` stands for "expression", but don't worry about this.

Suppose `var_num` is a numeric variable of some dataset.

```
gen var_total = total(var_num)
```

This will give an error, because `total()` is a function that must be used with `egen` if generating a new variable.

This will work:

```
egen var_total = total(var_num)
```

#### 9.4.2 anycount(varlist), values(intlist)

This function will search the variables inputted as a list (`varlist`) into the argument of `anycount`, for any integer value given in `intlist`. Suppose we have the following dataset:

```
var1    var2    var3
0       14      0
20      10      0
```

We want to count how many times the value 10 appears in each observation in any of the three variables, `var1`, `var2`, and `var3`.

```
egen var_count = anycount(var1 var2 var3), values(10 20)
```

Now, the dataset will look like this:

```
var1    var2    var3    var_count
0       14      0       0
20      10      0       2
```

In observation #1, the function found 0 instances of the value 10 *or* 20 in `var1`, `var2`, and `var3`. In observation #10, the function found 2 instances of the match of either 10 or 20.

#### 9.4.3 anymatch(varlist), values(intlist)

Instead of giving the count of how many times the values in `intlist` appears in the variables of `varlist`, this function will simply return 0 if that observation did not have any `intlist` values in `varlist`, and it will return 1 if a match was found.

Remember that to generate a new variable with the function, you must use `egen`.

#### 9.4.4 anyvalue(varname), values(intlist)

This function will return the value in `intlist` if found in the variable `varname`, and will return a missing value if not found.

Consider the following dataset.

```
var1
0
10
20
```

We generate a new variable using `anyvalue()`.

```
egen var_new = anyvalue(var1), values(10 20)
```

Now, the dataset is:

```
var1    var_new
0       .
10      10
20      20
```

For observation #1, the function did not find a match in `var1` with the value 10 nor 20. For observation #2, the function found a match with the 10 in `var1`, so it returns 10. For observation #3, the function found a match with the 20 in `var1`, so it returns 20.

#### 9.4.5 concat(varlist)

This function will concatenate the strings in each variable in `varlist`, by observation.

#### 9.4.6 count(exp)

This function counts the number of non-missing values in the variable `exp`.

#### 9.4.7 cut(varname), at(#,...,#)

This function returns a categorical variable, whose categories are formed by cutting the values of `varname` at the values given by `#,...,#`.

#### 9.4.8 diff(varlist)

This function checks, observation by observation, if there are any differences in the values of `varlist` variables. It returns 1 for observations that have differences between variable values, and 0 if all values across variables are the same.

#### 9.4.9 ends(varname)

This function returns the first word, bounded by whitespace, of each value in `varname`. If you want to extract the last word, add the `, last` option.

#### 9.4.10 fill(numlist)

`numlist` is a series of numbers, and this function detects and extends any patterns in this list. For example, if `numlist` is `1, 2, 3`, then `fill()` will extend the pattern of incrementing by 1 to all the observations for which `egen` is used.

#### 9.4.11 iqr(exp)

This function computes the interquartile range of `exp`. The interquartile range is the value of the 75% quantile of the data minuts the 25% quantile.

#### 9.4.12 kurt(varname)

This function computes the kurtosis of `varname`.

#### 9.4.13 skew(varname)

This function computes the skewness of `varname`.

#### 9.4.14 mean(exp)

This function computes the mean of `exp`.

#### 9.4.15 median(exp)

This function computes the median of `exp`.

#### 9.4.16 sd(exp)

This function computes the standard deviation of `exp`.

#### 9.4.17 mad(exp)

This function computes the median absolute deviation of `exp`. The median absolute deviation is the median of the absolute value of the sample's deviations from the sample's median.

#### 9.4.18 max(exp)

This function returns the maximum value in exp.

#### 9.4.19 min(exp)

This function returns the minimum value in exp.

#### 9.4.20 mdev(exp)

This function computes the mean absolute deviation of exp. The mean absolute deviation is the mean of the absolute value of the sample's deviations from the sample's mean.

#### 9.4.21 mode(exp)

This function returns the mode of exp, which is the most frequent unique value.

#### 9.4.22 pc(exp)

This function scales exp to a percentage of the total sum, which is between 0 and 100. If we add the , prop option, the proportion will be returned rather than the percentage. The proportion is between 0 and 1.

#### 9.4.23 pctile(exp)

This function computes the percentile, which is the quantile expressed as a percentage, of each observation within the entire variable. You can add the , p(n) option after the pctile(exp) portion in order to calculate the exact n-th percentile of exp.

#### 9.4.24 rank(exp)

This function assigns ranks to the values in exp. The largest value in exp is given rank 1, the second-largest value in exp is given rank 2, etc. By default, equal observations are assigned their average rank, but if you add the , field option, equal observations are given the same rank.

#### 9.4.25 row...(varlist) Functions

There are many functions with the prefix row..., such as rowmean(varlist), rowmax(varlist), or rowsd(varlist). These functions will compute the appropriate row-wise statistic for each observation, using only the variables supplied in varlist. For detailed descriptions of these functions, check the help egen documentation.

#### 9.4.26 std(exp)

This function maintains the same distribution that exp has, but standardizes the values of exp. It does so by scaling the mean to be 0 and the standard deviation to be 1 (by default), but again, it does not change the probability distribution.

#### 9.4.27 tag(varlist)

This function is a tool to mark the position at which consecutive observations have different values in varlist. Whenever the variable in varlist changes value, a 1 is used to mark that observation. 0s are placed in all other observations. For example, consider the following dataset.

```
var1
0
0
0
0
```

```
0
1
1
1
2
2
2
2
```

We tag the positions where `var1` changes values, generating a new variable to hold these markings.

```
egen markings = tag(var1)
```

The dataset is now as follows.

```
var1    markings
0       1
0       0
0       0
0       0
0       0
1       1
1       0
1       0
2       0
2       0
2       0
2       0
```

# 10 Exercises

1. Load any example dataset (*File > Example datasets...*). Use `codebook` on a 1 variable, `inspect` on all variables, and `summarize` a numerical variable.
2. Load the S&P 500 example dataset.
   a) List the `open` variable if higher than the average. How many results are there?
   b) List the `close` variable if higher than the `high` variable. How many results are there?
   c) List the `open` variable if the `change` variable is higher than zero. How many results are there?
3. Load the `auto` example dataset.
   a) Save the dataset twice with with the names `dataset1.dta` and `dataset2.dta`.
   b) Append `dataset1` under `dataset2`.
   c) How many observations does the resulting dataset have?
4. Load the `census` example dataset.
   a) Keep only the `state` and `pop` variables, sort by `state`, and save the dataset as `data1.dta`.
   b) Open the `census` dataset again, keep only the `state` and `marriage` variables, sort by `state`, and save the dataset as `data2.dta`.
   c) Merge `data1` with `data2`.
      i) Open `data1.dta`.
      ii) Merge it with `data2.dta`.
      iii) Analyze `_merge` new variable.
5. In this exercise, we make a database from scratch.
   a) Set 1000 observations in a new dataset.
   b) Create a variable called `random` using the `random` function.
   c) Copy the previous variable into a new variable called `random_modified`.
      i) Replace all values lower than 0.25 with 0.
      ii) Replace all values higher than 0.75 with 2.
      iii) Replace all values in between 0.25 and 0.27 with 1.
   d) Save the dataset with any name.
6. Use the dataset made in Exercise 5. Drop the second variable, and rename the `random` variable to `random_changed`. Rename `random_modified` to `random_modified2`.
7. Load the `census` example dataset. Sort by `region` and `pop`. Think about what should have happened to the dataset with this command, and verify by opening the data editor.
8. Load the `census` example dataset. Count any variable, and verify this result yourself. Count the `pop` variable if `pop > mean(pop)`. Count the `state` variable if `region` is equal to "West" (4).
9. Use the S&P 500 example dataset.
   a) Find the observations for which `open` was higher than `close`. (Hint: Use `gen var = (open > close)`. Open the data editor. What happened?)
   b) Find the observations for which `open` was equal to `close`. (Hint: Use `gen var = (open == close)`. Open the data editor. What happened?)
10. Use the S&P 500 example dataset.
    a) Create a new variable called `open_lagged10`, which is equal to the `open` variable but lagged 10 periods.
    b) Create a new variable called `open_forward5`, which is equal to the `open` variable but forwarded 5 periods.
    c) Open the data editor and check your results.

11. Use the S&P 500 example dataset.

    a) Create a new variable called `open_dif`, which is equal to the `open` variable but differenced 1 period. Is `open_dif` the same as the `change` variable? Why?
    b) Create a new variable called `open_seasonal`, which is equal to the `open` variable but seasonal 1 period. Is `open_seasonal` the same as the `change` variable? Why?
    c) Create a new variable called `open_seasonal2`, which is equal to the `open` variable but seasonal 2 periods. Is `open_seasonal2` the same as the `change` variable? Why?

12. Load the `auto` example dataset.

    a) Display the average of the `price` variable by summarizing `price` and displaying `r(mean)`.
    b) Did it display the same value as the mean of `price`?

13. Load the `auto` example dataset. Create an indicator variable for the `make` column, by performing the following:

    a) Go to indicator variable creation and select the `make` column. Press OK.
    b) How many different columns would you expect STATA to create for the indicator variable of `make`?
    c) Verify your answer to part (b) by opening the dataset.

14. Load the `auto` example dataset.

    a) Display the average of the `price` variable.
    b) Display the average of the `price` variable using the `turn` variable as frequency weights. Why are the results different?
    c) Display the average of the `price` variable using the `turn` variable as sampling weights.

15. Load the `auto` example dataset.

    a) Apply `destring` to the `foreign` variable. What message did you get? Why?
    b) Apply `tostring` to the `make` variable. What message did you get? Why?

16. Learning STATA syntax will: (Choose one.)

    - Not help me at all
    - Help me create only graphs
    - Help me create only descriptive statistics
    - Help me create only statistical estimations
    - Help me do anything inside STATA

17. True or False: Creating do-files not only lets you save lots of statistical code, but also lets you share code among friends and/or co-workers.

18. True or False: STATA's help PDF documentation is so vast that you can see methods and formulas for all STATA commands

19. True or False: I can see the inner calculations of all STATA commands in the Remarks and Formulas section of the STATA manuals.

20. True or False: Regarding the summarize command, I can either type "`summarize`" or "`summar`" to get the same result.

21. True or False: I need to click on the command line prompt ever time I want to type some command.

22. True or False: I can type abbreviated commands and variable names.

23. Which command is used to allocate memory in STATA for database reading? (Choose one.)

    - `allocate memory`
    - `ready memory`
    - `build memory`
    - As of newer STATA versions, there is no need for memory allocation anymore.

24. True or False: Given STATA's new memory allocation functionality for databases, it is a good practice to use the `compress` command from time to time.

25. True or False: If I change the working directory, all results will be directed to that new directory.

26. True or False: I can do statistical testing using pre-installed STATA databases.

27. True or False: If I "`clear`" a database, it will no longer be in STATA's current memory.

28. True or False: I can view a database by either the `browse` or `edit` commands.

29. What is the point of the `list` command if I already have the `browse` and `edit` commands? (Choose one.)

    - There is no point at all.
    - It can be used with "`if`" and "`in`" expressions, giving more flexibility to view data in the results window.
    - It lists the variable names and formats of the current database in memory.
    - It lists the history of all commands used in the current STATA session.

30. If I open a database and append 5 different datasets to it, STATA will: (Choose one.)

    - Paste at the bottom only the biggest dataset
    - Paste at the bottom each dataset one at a time, until all 5 datasets are appended
    - Paste at the right each dataset one at a time, until all 5 datasets are appended
    - Do nothing, as the `append` command does not exist

31. True or False: Since STATA does not know how to merge different datasets together, both datasets have to have a unique identifying variable that will serve as the merging criterion.

32. True or False: If I have a monthly database and I merge it with a daily database, then, if done correctly, monthly values will be "repeated" in order to give space for daily values.

33. True or False: A very good practice is to label both the database and its variables.

34. True or False: Almost all STATA estimations require the database to be in long format.

35. True or False: In transposing a database, the variable names will be edited.

36. If I have a database with daily observations, but I want to convert it to weekly means, then I'd use the command: (Choose one.)

    - `bysort week: contract`
    - `bysort week: collapse`
    - Cannot be done in STATA
    - `bysort week: means`

37. True or False: If I want to create a database from scratch, I should start by setting the number of observations with the `set obs` command.

38. True or False: A very quick way to copy a variable is to generate a new variable equal to the original, like: `gen var2 = var1`.

39. True or False: A quick way to view variable labels is by using the `describe` command.

40. Suppose I have two variables: `sales` and `year`. If I want to replace all sales for 2015 with 0, I'd use the following command: (Choose one.)

    - `replace sales = 0 in year == 2015`
    - `replace sales = 0 if year == 2015`
    - `replace sales = 0 if year = 2015`
    - `replace sales == 0 if year == 2015`

41. Suppose I have the following variables: `year`, `month`, `day`, and `sales`. To sort the entire dataset in descending order (newest first), I should use the following command:

    - `sort year month day`
    - `gsort -year -month -day`
    - `sort -year -month -day`
    - `sort -year month -day`

42. True or False: Every time a new variable is created, STATA will place it at the far right of the current database. So, a good command to bring it to the far left is `order`.

43. Suppose I have the variables `pets`, `zipcode`, and `gender`, and assume that `pets` has no missing values. If I want to count how many male-owned pets there are in zipcode 33122, I should use the command: (Choose one.)

    - `count where zipcode = "33122" and gender = male`
    - `count in zipcode = 33122 and gender = "male"`
    - `count if zipcode == 33122 & gender == "male"`
    - `count to zipcode == 33122 | gender = "male"`

44. Suppose I have a `sales` variable, and we have declared our dataset as a time series with a `year` variable. The first difference of sales is _____ and its third seasonal variant is _____. (Choose one.)

    - `L.sales`; `F.sales`
    - `D.sales`; `L.sales`
    - `D.sales`; `S3.sales`
    - `L2.sales`; `F2.sales`

45. If I have a `sales` variable and I want to generate the natural log of the square of `sales`, then I should use the command: (Choose one.)

    - `gen new_var = log(sales^0.5)`
    - `gen new_var = ln(sales^0.5)`
    - `gen new_var = ln(sales^3)`
    - `gen new_var = ln(sales^2)`

46. True or False: Every time I lag a variable, I will lose an observation.

47. If I want to "get" a result (yellow colored by default), then I'd need its stored result name. This can be readily searched for in: (Choose one.)

    - `help` "command" and in the description section
    - `help` "command" and in the options section
    - `help` "command" and in the syntax section
    - `help` "command" and in the stored results section

48. Suppose I have an `age` variable. If I want to create all necessary variables to identify each age, then I'd type: (Choose one.)

    - `tab age`
    - `tab age, do(new_)`
    - `tab age, gen(new_)`
    - `table age, gen(new_)`

49. Suppose I have the variables `gender` and `smoke_yes_no`. If I want to identify all interactions between those two variables, then I'd type: (Choose one.)

    - `xi gender * smoke_yes_no, sufix(new_)`
    - `x gender * smoke_yes_no, prefix(new_)`
    - `xi gender * smoke_yes_no, prefix(new_)`
    - `xi gender smoke_yes_no, prefix(new_)`

50. True or False: I can use any stored results, as long as they are yellow colored in the results window in STATA.

51. Supose I have a `sales` variable and an `area` (in square miles) variable. If I want to compute descriptive statistics of `sales` weighted by area, then I type: (Choose one.)

    - `summ sales [fweight = area]`
    - `summ sales [pweight = area]`
    - `summ sales [aweight = area]`
    - `summ sales [iweight = area]`

52. Supose I have a `sales` variable and a `zipcode` variable with 3 different zipcodes inside, and the database is 300 observations long. If I want to compute descriptive statistics for `sales` split by all zipcodes, a quick way is: (Choose one.)

    - `summ sales if zipcode == 1; summ sales if zipcode == 2; summ sales if zipcode == 3`
    - `bysort zipcode: summ sales`
    - `through zipcode: summ sales`
    - `bysort zipcode: describe sales`

53. Suppose I imported a database, a process in which a variable `sales` was imported as text. Assuming that `sales` has no values other than numbers inside, I can transform it into a numerical variable by typing: (Choose one.)

    - `real(sales)`
    - `destring sales, replace`
    - `tostring sales, replace`
    - `transform sales`

54. True or False: A quick way to check whether some command is applicable with weights is to access its help file in STATA.

55. True or False: In order to set two different graphs side-by-side in one plot, I must overlay them.

56. True or False: I can overlay as many graphs as I want.

57. True or False: In order to combine graphs, I have to save them separately first, then call them by their saved names with the `combine` command.

58. True or False: A quick way to create the "perfect" graph is to use the corresponding dialog box and hitting the "submit" button.

59. If I wanted to compute the variation coefficient (standard deviation divided by the mean) for the `sales` variable, then after summarizing `sales`, I could type: (Choose one.)

    - It's not possible to deduce the variation coefficient after summarizing `sales`, because this is not part of the results shown.
    - `display r(sd)/r(mean)`

60. A good way to quickly inspect the contents of any numerical variable is to use the `tabulate` command, unless: (Choose one.)

    - The variable has a large sample and is continuous.
    - The variable contains non-numeric characters.
    - The variable has a small sample.
    - The variable has a lot of missing values.

61. True or False: There is no way to obtain the sorted frequency distribution of a categorical variable.

62. STATA computes confidence intervals for the mean using the t-distribution (as outlined in the PDF help documentation's "remarks" section) instead of the normal distribution, because: (Choose one.)

    - The sample is large.
    - The variable is continuous and in logs.
    - STATA assumes the variance is unknown and estimated.
    - This is an error with STATA.

63. True or False: Even if STATA uses the t-distribution for computing confidence intervals for the mean, the results will converge if the sample is large.

64. Suppose we already know from a scatter graph that the continuous variables `var1` and `var2` have a "U"-shaped association. Then, the correlation coefficient will be: (Choose one.)

    - Greater than 0 and less than 1
    - 0
    - Less than 0 and greater than -1
    - Inappropriate to use for this problem, regardless of its value, because STATA computes the Pearson correlation coefficient, which measures the linear degree of association between two continuous variables

65. True or False: The math functions `int()` and `floor()` will have different results only when the variable is already an integer.

66. True or False: `mod(8,3)` = 2.

67. If `sign(1)` = 1 and `sign(-1)` = -1, then `sign(0)` is: (Choose one.)

    - 0.5
    - 0
    - 1
    - 0.1

68. `gen X = sum(var1)` gives inside X: (Choose one.)

    - The cumulative sum in the order that `var1` is sorted
    - The total sum of `var1` repeated in all cells
    - The sum of `var1` and `var2`
    - The sum without decimals

69. If I have a string variable that actually contains numbers, we can use the following command to convert it into a numerical variable: (Choose one.)

    - `regexm()`
    - `real()`
    - `strcat()`
    - `substr()`

70. If I have a `name` variable in which all names start with "Mr." (e.g. "Mr. David Smith"), and I want to create a new variable without the prefix, then I could use the command: (Choose one.)

    - `gen name_without_prefix = substr(name, 3, 10)`
    - `gen name_without_prefix = substr(name, 1, 1)`
    - `gen name_without_prefix = substr(name, 4, .)`
    - `gen name_without_prefix = substr(name, 10, 4)`

71. True or False: Since `strlower("DOG")` = "dog", `strlower("ENTRÉE")` = entrée.

72. True or False: `strtrim("This is my name: Raul")` = "Thisismyname:Raul"

73. If I have a variable `var_with_Mr`, which contains names starting with "Mr.", and I want the prefix to be replaced with "Mister" in a new variable called `var_with_Mister`, then I'd use the command: (Choose one.)

    - `gen var_with_Mister = subinstr(var_with_Mr, "Mr.", "Mister", 1)`
    - `gen var_with_Mister = subinstr(var_with_Mr, "Mister.", "Mr", 1)`
    - `gen var_with_Mister = subinstr("Mister", var_with_Mr, "Mr.", 1)`
    - `gen var_with_Mister = subinstr(1, var_with_Mr, "Mr.", "Mister")`

74. True or False: To generate random numbers that do not follow any probability distribution, I should use the uniform random number function.

75. True or False: Any random number function starts implicitly with a "seed" number, which can be manually set according to the researcher's requirements.

76. If I need to quickly generate some random numbers that follow a normal (bell-shaped) distribution, then I can use the function: (Choose one.)

    - `rbeta()`
    - `rt()`
    - `rnormal()`
    - `rchi2()`

77. When using the `summarize` command, I get column-wise results, such as the mean and standard deviation for a variable. If I want to get row-wise results for the mean and standard deviation across several variables, I should use the following `egen` commands: (Choose one.)

    - `anycount()`, `anyvalue()`
    - `total()`, `pctile()`
    - `mean()`, `sd()`
    - `rowmean()`, `rowsd()`

78. The relationship between `sum()` and `total()` is that: (Choose one.)

    - `sum()` will compute the cumulative sum by row, whereas `total()` will show the same total sum in every row.
    - `total()` will compute the cumulative sum by row, whereas `sum()` will show the same total sum in every row
    - They are exactly the same

79. If I use the `egen` function `kurt()`, I will have a new variable with the result "copied" in every row of the current dataset. So, a quick way to "get" that result is: (Choose one.)

    - `display new_var in 1`
    - `display new_var with 1`
    - `display new_var if 1`
    - `display new_var & 1`

## 10.1 Solutions to Selected Exercises

16. Help me do anything inside STATA
17. True
18. True
19. True
20. True
21. False
22. True
23. As of newer STATA versions, there is no need for memory allocation anymore.
24. True
25. True
26. True
27. True
28. True
29. It can be used with "`if`" and "`in`" expressions, giving more flexibility to view data in the results window.
30. Paste at the bottom each dataset one at a time, until all 5 datasets are appended
31. True
32. True
33. True
34. True
35. True
36. `bysort week: collapse`
37. True
38. True
39. True
40. `replace sales = 0 if year == 2015`
41. `gsort -year -month -day`
42. True
43. 'count if zipcode == 33122 & gender == "male"
44. `D.sales`; `S3.sales`
45. `gen new_var = ln(sales^2)`
46. True
47. `help` "command" and in the stored results section
48. `tab age, gen(new_)`
49. `xi gender * smoke_yes_no, prefix(new_)`
50. True
51. `summ sales [fweight = area]`
52. `bysort zipcode: summ sales`
53. `destring sales, replace`
54. True
55. False
56. True
57. True
58. True
59. `display r(sd)/r(mean)`
60. The variable has a large sample and is continuous.
61. False
62. STATA assumes the variance is unknown and estimated.
63. True
64. Inappropriate to use for this problem, because STATA computes the Pearson correlation coefficient, which measures the linear degree of association between two continuous variables
65. True
66. True

67. 0
68. The cumulative sum in the order that `var1` is sorted
69. `real()`
70. `gen name_without_prefix = substr(name, 4, .)`
71. False
72. False
73. `gen var_with_Mister = subinstr(var_with_Mr, "Mr.", "Mister", 1)`
74. True
75. True
76. `rnormal()`
77. `rowmean()`, `rowsd()`
78. `sum()` will compute the cumulative sum by row, whereas `total()` will show the same total sum in every row.
79. `display new_var in 1`

# 11 Acknowledgements

This course is based off of Udemy's *Complete STATA Workflow + Tips* by Mauricio Maroto. Stata 16 documentation was also essential in the writing of these notes.