Introductions

Introductions

- ***Cale*** Basaraba

  - Born and raised in NYC
  - Stuyvesant High School
  - Stanford University: B.A. Philosophy, B.S. Symbolic Systems
  - Mary McDowell Friends School: Teacher 2009-2014
  - Columbia University: MPH Epidemiology, Applied Biostatistics

Biostatistics & Programs

- During my MPH, I worked for Dr. Carolyn Westhoff as a biostatistician on the pharmacokinetics of oral contraceptives (ethinyl estradiol and levonorgestrel)
- Dr. Westhoff and Dr. Pike work with Stata, so I taught myself Stata
- Most of my coursework in biostatistics and epidemiology used SAS
- I used R in my thesis work and in my current position

EcoHealth Alliance

Local conservation.
Global health.

- Research Scientist at EcoHealth Alliance on the Modeling & Analytics Team
  - NGO here in midtown that works on pandemic prevention through building an understanding of viral spillover from animals
  - In my day-to-day, I work on epidemiological compartmental models, computer programming, data visualization, and hierarchical modeling

Who are you?

- If possible, could we do a brief introduction around the room:
  - Name
  - Interests / Field
  - Current Preferred Program for Data / Statistics

Workshop Goals

1. Learn a broad range of basic Stata skills

2. Become *comfortable* with Stata

3. **Acquire the tools and resources to independently learn more about Stata**

Switching Programs

- Many researchers are loath to switch from one programming language to another
- This is reasonable, because they have built up years of expertise in one language
- However, we tend to remember how hard it was to learn our **first** programming language, and don't realize how many of the skills can be translated
- The programming concepts, data structures, and even commands are often similar between languages

Learning Stata

- The learning curve is not as steep as other programs
- Learning in a group workshop environment can get you through the worst of it
    - Ask lots of questions — to me and to your colleagues
    - Experiment (you can't break anything!)
- At first, all the commands and the syntax can be daunting, but there is **no reason to memorize specifics**
- Focus on the major concepts and remembering *where to find the specifics, not the specifics themselves*

Workshop Structure

- At the beginning of each workshop, I will go through slides and **live coding** examples
    - **Live coding** means I will make mistakes — help me catch them!
    - **Live coding** also means sometimes I'll have to pause to think about my mistakes
    - Use your sticky notes to indicate that you're having trouble
        - If your neighbor needs help, lend a hand!

Workshop Structure

- After going through the concepts and showing some live coding examples, there will be a few exercises for you to try on your own
- Working in groups is a great way to learn and pool beginner knowledge
- Towards the end of the period we can code solutions together

Stata Overview, Interface, and File Types

Stata

- Stata advertises itself as a fast, easy-to-use, and powerful statistical software package
- True! In my humble opinion, Stata is very fast, one of the easiest package to learn, and has support for a wide (and growing) array of analyses
- Stata also prides itself on being both a point-and-click **and** command-line based package
    - *This workshop is all about the command line*

Stata Pros / Cons

PROS

- Fast, Strong, Efficient
- Intuitive (for the most part)
- Well-organized and understandable documentation
- Proprietary (Unified Vision, Overarching Plan)
- Reproducible
- Publication-level graphics

CONS

- Proprietary (Licenses, Top-down control)
- Idiosyncratic Syntax
- Less flexible for bespoke analyses
- Proprietary != External Reproducibility
- Smaller User Ecosystem
- Less flexible data management
- Too many windows!

Stata = Great!

- Take my opinions with a grain of salt — Stata is a **GREAT** option for organizations trying to improve internal reproducibility
- Let's all take a look at the Stata Interface together:
  - Main Window
    - Command (*where you type commands)*
    - Results *(where you view results)*
    - Review *(a history of commands you have run)*
    - Variables *(a quick look at the variables in your dataset)*

Datasets

- In Stata, a dataset is a group of observations with values for a group of variables, along with metadata about the data, the variables, and what their levels mean (in the form of labels and notes)

- A Stata dataset file is called a **dta-file** and has the file extension **.dta**

- Stata can read many other types of data files, but works best with **.dta** files because of its rich metadata tracking system

Do File Editor

- Stata's power gets unlocked when you combine a series of commands together into what's called a **do-file**.
    - In other languages this might be called a *script*, but in Stata it's a do file.
    - These files are Stata-specific and have the extension .do (e.g. `clean_data.do`)
    - Let's run the `setup.do` file to download some files we're going to use!

Data Browser

- Stata has a built-in Data Browser which allows you to peruse your data set
    - This shows your data in a traditional excel-file type spreadsheet format, but presents categorical variable labels in a useful way
- There is also a Data Editor for manually changing values in the spreadsheet
    - **I strongly advise against using this!**

Viewer Window

- Whenever you ask for `help` with a command or concept, the Viewer Window will pop up, showing you Stata's built-in documentation

- At the top of these help files there are often links to even more detailed Stata manuals (in PDF form).

Graph Editor

- The Graph Editor is a point and click tool for editing graphs
- It allows the user to customize almost every part of a graph that has already been created (but the graph must be created first to manipulate)
- Stata graphs can be saved with the **.gph** extension. They are not flat files (like a jpeg or a tif), they can be loaded and re-manipulated
- **Point-and-click solutions work counter to automation and reproducibility**
    - However, sometimes you want to just change that one little thing…

Windows Galore!

- Stata has a lot of distinct windows — my advice is to mainly write scripts and run commands from the Do File Editor
- I keep the Main Window open on one side of my screen, with the Do File Editor on the other
- I tend not to use the Data Browser often, but your preferences will evolve as you work more with Stata

Loading Data and an Introduction to Variables and Labels

Stata Data

- When Stata begins, no data is loaded into memory, and the Variables, Data Editor, and Data Browser windows will all be empty

- Stata works with one dataset at a time. In order to examine other datasets the user must explicitly switch between them

- Stata has built-in protections to help users avoid accidentally **`clear`**-ing unsaved data from memory or **`save`**-ing over files on the hard drive

Loading Data

- The most streamlined way of loading data into Stata is by loading a Stata `.dta` file

- In this case, you simply use the command:

```
use filename
```

- Stata comes with toy datasets already loaded for teaching purposes, as well as easily accessible web-based datasets

```
sysuse dataset
```

```
webuse dataset
```

Excel Files

- One of the many positive features of Stata is its strong Excel file support (**.xls** and **.xlsx** files)

```
import excel filename, firstrow
```

- This command will load the Excel file into Stata and use the first row to determine the variable names

- When possible, use the import excel command because it correctly deals with dates and number types

    - Without the `firstrow` option, the variable names will be set as Excel column letters (e.g. A, B, C, D)

Delimited Files

- Stata can easily import comma-separated (**.csv**) files or tab-separated (usually **.txt**) files using one command:

```
import delimited filename
```

- The import command can specify only certain rows or columns as well

```
import delimited filename, rowrange(start:end)
colrange(start:end)
```

- This is one of many options available. Use `help import` to browse through them all!

Fixed Format Files

- Stata can also import fixed-format files using dictionary files
    - These dictionary files must define the types for each column

```
infile using dictionary_file, using(data_file)
```

- For examples of this type of file loading, see **`help`** `infile fixed` (not a file type I am very familiar with)

Navigating Directories

- When loading any file, you need to make sure that Stata is pointing to the correct directory (where that file is present)
    - It is possible that your Stata session is currently "pointing" to a different folder on your computer
- To check where you currently are, use the command **`pwd`**
- To take a look at what is in your current directory you can use the **`ls`** command
- In order to change directories, you can use the **`cd`** command along with a folder in your current directory or a filepath

Variables

- The word “variable” can mean many things in programming and statistics, but in Stata it refers particularly to the social scientist’s common definition of a variable
    - In databases this might be called a *field,* or *column*. When using the Data Browser or issuing the command `list`, variables will be represented as *columns*
    - There are other “variables” held in memory and available for Stata users, but these remain hidden to the beginner

Observations

- In Stata, observations refer to the entries in your dataset, each of which has values for some, if not all, of your variables

- An observation is usually considered the unit of your analysis. Often, each observation is unique in some way, but for certain analyses complete duplicates may be present

- Stata observations make up the *rows* of your dataset when calling `list` or using the Data Browser

Basic Commands

- In order to learn some basic things about variables, we will be using some basic commands:

- Commands are actions performed on datasets

`generate` — generates a new variable

`replace` — replaces an already existing variable

`describe` — describes the details of a dataset or variable

`drop` — drops variables or observations (when combined with an if statement)

`keep` — keeps variables or observations (when combined with an if statement), the opposite of **`drop`**

generate

```
generate variable_name = expression
```

- This command generates a new variable based on a given expression
- The expression can be a constant (e.g. a number 4, or a string “Hello!”) or can be dependent on other variables (e.g. `weight/height` or `age + 5`).
- This command will only generate a new variable if that variable name is available

replace

```
replace variable_name = expression
```

- This command replaces the values of a pre-existing variable based on a given expression

- Just like **generate** this can be a constant or dependent on other variables (e.g. mpg*price).

describe

describe

describe variable_name

- This command will output a description of the variable, including the variable name, storage type, value label, and variable label

```
. describe rep78

              storage   display    value
variable name   type    format     label      variable label
------------------------------------------------------------------------------
rep78           int     %8.0g                 Repair Record 1978
```

drop

```
drop variable_name
```

- The **drop** command with `variable_names` will remove ***variable***(s) from the dataset

```
drop if rule
```

- The **drop** command with a `rule` will drop ***observations*** that satisfy that rule from the dataset

keep

- The **keep** command is the reverse of the **drop** command

```
keep variable_name
```

- The keep command with variable_names will keep only the ***variable***(s) specified in the dataset

```
keep if rule
```

- The keep command with a rule will keep only ***observations*** that satisfy that rule in the dataset

`if` rules

- This `if` is the `if` qualifier, not the `if` from a `if/else` programming statement
- `if` qualifies a command, telling Stata that we want to perform our action on only a subset of our data
- In the case of **`drop`** or **`keep`** commands, `if` tells Stata what observations to drop, or what observations to keep

```
keep if mpg > 25

drop if headroom < 2
```

if rules

```
keep if mpg > 25
```

The rule evaluates to a True (1) or False (0) for each observation. If it evaluates to True, the command is performed on that observation.

```
keep if headroom == 2
```

Rules that check equality use double equal signs. This == tells Stata you are checking equality, not assigning something.

if rules

```
keep if mpg >= 25 & headroom < 3
```

(mpg is greater or equal to 25 AND headroom is greater than 3)

```
keep if headroom != 2 | mpg > 30 | price < 80000
```

(headroom does not equal 2 or mpg is greater than 30 or price is less than 8000)

Rules can use boolean logic operators like *AND* and *OR.* In Stata the ampersand **&** represents AND, while the vertical line | represents OR. The entire statement will be evaluated as True (1) or False (0).

Rules that check equality use double equal signs. This == tells Stata you are **checking for equality**, *not assigning something*. The != symbol combination means **not equal**.

Labels

- Labels are a built-in way of conveying metadata about your variables and dataset to others and to yourself
- There are several labeling scopes in Stata (dataset, variable, variable levels (i.e. values))
- For now, we will take a look at dataset and variable labels

Dataset and Variable Labels

label data *label*

- This command will label your entire dataset with a particular *label* that can be seen when using the command **describe dataset_name**

label variable **variable_name** *label*

- This command will assign a *label* to a variable in your dataset that can be seen when using the command **describe variable_name** or by looking in the variables portion of your Stata window

Exercises (1)

1. Titanic Data

    A. Create a do file called titanic.do

    B. Load the titanic.csv file into Stata

    C. Drop all variables except name, age, sex, survived, and fare

    D. Only keep observations whose age is greater than 20 and who survived

    E. Try your best to give informative labels to each variable

Exercises (2)

2. Movie Metadata

   A. Create a do file called movies.do

   B. Load the movie_metadata.xls into Stata (remember the **firstrow** option)

   C. Drop all movies with runtimes equal to or less than 45 minutes

   D. Keep the variables: duration, gross, movie_title, country, budget, and imdb_score

   E. Limit the dataset to only movies from the United States

Variables: Types, Generation, Replacing, Labeling, Recoding, Notes

Variables

- The word “variable” can mean many things in programming and statistics, but it has a specific meaning in Stata:
    - A variable in Stata reflects the social scientist’s common definition of a variable, not the programmer’s
    - In databases this might be called a *field,* while in other languages it might be simply a *column*
    - There are “variables” held in memory and available for Stata users, but these remain hidden to the beginning user

Basic Commands

- In order to learn some basic things about variables, we will be using three commands:

`generate` — generates a new variable

`replace` — replaces an already existing variable

`describe` — describes the details of a variable

Generate

```
generate [type] variable_name = expression
```

- This command generates a new variable with type *type* based on a given expression
- The expression can be a constant (e.g. a number 4, or a string “Hello!”) or can be dependent on other variables (e.g. weight/height or age + 5).
- This command will only generate a new variable if that variable name is available

Replace

```
replace variable_name = expression
```

- This command replaces the values of a pre-existing variable based on a given expression
- Just like `generate` this can be a constant or dependent on other variables (e.g. weight/height or age + 5).
- This command will only generate a new variable if that variable name is available

Describe

```
describe
```

```
describe variable_name
```

- This command will output a description of the variable, including the variable name, storage type, value label, and variable label

```
. describe rep78

              storage   display    value
variable name   type    format     label      variable label
------------------------------------------------------------------------------
rep78           int     %8.0g                 Repair Record 1978
```

Variable Types

- Every Stata variable will be defined as a particular *type*
  - The *type* of a variable defines what kind of data is expected for each variable (a number or a string of characters)
  - The *type* of a variable also defines what kinds of commands can be performed on a variable, and what kind of results we can expect from these commands
  - Finally, the *storage type* of a variable determines how the variable is stored in memory (more important for advanced users)

Types and Missing

- Number: Height measured in inches
  - A missing number will be represented with a `.`
  - To see entries with missing values, you can use the command `list if missing(variable_name)`
- String: A participant's name or address
  - Missing strings are represented by the empty string `""`

Example: Numbers

- Numbers come in 5 storage types:
  - byte
  - int
  - long
  - float
  - double

Storage type	Minimum	Maximum	Closest to 0 without being 0	Bytes
`byte`	-127	100	± 1	1
`int`	$-32{,}767$	32,740	± 1	2
`long`	$-2{,}147{,}483{,}647$	2,147,483,620	± 1	4
`float`	$-1.70141173319 \times 10^{38}$	$1.70141173319 \times 10^{38}$	$\pm 10^{-38}$	4
`double`	$-8.9884656743 \times 10^{307}$	$+8.9884656743 \times 10^{307}$	$\pm 10^{-323}$	8

if rules and .

- There is one important Stata idiosyncrasy regarding if qualifiers and missing numbers
    - Behind the scenes, Stata records a missing `.` as a *very large number*
    - As a result, any if rule using a `>` or `>=` will always evaluate to True (1) when a missing is present

```
generate m = .

generate test = 1 if m > 23
```

- Whenever you are using these operators, it makes sense to check beforehand for missing values, or include an additional clause to your rule:

```
generate test = 1 if m > 23 & !missing(m)
```

Number Storage Types

- Storage types are not a big concern for beginning users
  - Stata is very intelligent about default variable type creation, and expanding types when using the `replace` command
  - However, incorrect assignment of types using the `generate` command can result in missing values **without** Stata reporting an error

Keep It Simple!

- For the beginning user, messing around with types can lead to errors
- But an understanding of what types are can sometimes help you figure out odd behaviors and missing data in variables
- It only makes sense to call certain commands on certain types of variables (calling `mean` on a string variable does not make sense)

Labels

- There are several labeling scopes in Stata (dataset, variable, variable levels [i.e. values])
- Labels are a very useful way of conveying metadata about your variables and dataset to others and to yourself
- Good practice suggests that variables and levels of categorical variables should always be labeled
    - Without proper labeling and coding, the interpretation and manipulation of unfamiliar datasets can be needlessly difficult

Dataset and Variable Labels

label data *label*

- This command will label your entire dataset with a particular *label* that can be seen when using the command **describe dataset_name**

label variable **variable_name** *label*

- This command will assign a *label* to a variable in your dataset that can be seen when using the command **describe variable_name** or by looking in the variables portion of your Stata window

Variable Levels

- Categorical variables are very common in the social sciences, but must be labeled well to avoid misinterpretation
- Like many statistical programming languages, Stata analyzes categorical variables by assigning numerical values to categories (e.g. “Living” = 0, “Deceased” = 1).
- The actual values assigned to dichotomous variables are arbitrary and only relevant for interpretation (but it makes sense for 0/1 to match the variable name)
- Depending on the type of modeling, dummy variables or ordinal variables will require certain values

Generate a Categorical Variable with Labels

- Generate the numerical values based on another variable

```
generate eff_car = 1 if mpg > 33

replace eff_car = 0 if mpg <= 33

replace eff_car = -1 if mpg < 20
```

- Define a label for each value

```
label define label_name level1 “Label1” level2
“Label2”

label define eff_car_label 1 “Great” 0 “Fair” -1
“Poor”
```

More Label Details

- Assign label to variable

```
label values variable_name label_name
```

```
label values eff_car eff_car_label
```

- Modify existing label

```
label define label_name level “newLabel”, modify
```

```
label define eff_car_label 0 “Good”, modify
```

`label` and `codebook`

```
label list
```

- This command will output a neat list of all the labels in the current dataset

```
label dir
```

- This command will output the variables with labels

```
codebook
```

- This command will output a detailed codebook with information

```
codebook, problems
```

- This command highlights potential problems with current dataset (variables with more than 9 values are assumed to be continuous)

encode

```
encode variable_name, generate(new_variable)
```

- This command creates a new **number** variable from a string variable, with level labels that correspond to the original string variable
- This is a quick and easy way to generate correctly labeled numerical coding of string variables
- We can give this a try for the countries in our movie_metadata file.

```
encode country, generate(country_code)

codebook country_code
```

recode

- The `recode` command is an easy way to recode existing **numerical variables** base on a simple rule

```
recode variable_name rule
```

- For example, we could recode the -1 in the `eff_car` variable (saving first):

```
save autotemp
```

```
recode eff_car -1 = 2
```

recode

- Often you want to recode more than one number, to do so you can separate your rules using parentheses:

```
use autotemp, clear

recode eff_car (-1 = 0) (0 = 1) (1 = 2)
```

- Additionally, you can recode and generate a new variable rather than replacing your current one:

```
use autotemp, clear

recode eff_car (-1 = 0) (0 = 1) (1 = 2), generate(new_eff_car)
```

- Finally, you can combine this with the immediate creation of new level labels:

```
recode eff_car (-1 = 0 “Decent”) (0 = 1 “Improved”) (1 = 2
“Excellent”), generate(new_eff_car) label(new_eff_car)
```

Notes

- The **`notes`** command allows users to add notes to a dataset or a variable that will be permanently linked when the data is saved

- You might want to include details on data provenance, reminders to collaborators, or questions about a particular value

`notes` - displays all notes for a dataset

`notes``:`*`note`* - adds a note to the dataset as a whole

`notes` `variable_name:`*`note`* - adds a note to a particular variable

Exercises (1)

1. Auto Data

    A. Create a do file called auto.do

    B. Call **sysuse** auto (to load auto data)

    C. Create a new price category variable (price_cat). If a car is less than 4000, assign a 0; between 4000 and 6000, assign a 1; greater than 6000 assign a 2.

    D. Give your new variable a thoughtful label. Then, create custom labels for these three values of price_cat and assign them.

    E. Modify the most expensive label to be “Fancy”

    F. Add a note to this dataset with your name and today’s date

Exercises (2)

1. Movie Metadata

    A. Edit your movies.do file so that your dataset will include movies from all countries – then run it.

    B. Create a new categorical variable (country_code) for countries. Give this variable a meaningful label

    C. Use the codebook to check how many observations have missing country_codes, then drop these observations from the dataset.

    D. Create a new categorical variable (cheap). Movies with a budget over 100,000,000 should have a 0, others have 1. Watch out for missing values!

    E. Rename the cheap variable to expensive. Recode it so that observations that used to be 0 are 1, and vice versa

    F. Add brief notes to these two new variables with your name and date

Introduction to Commands, Basic Descriptive Statistics

Stata Commands

- Stata is organized around built-in commands
- Commands are “verbs” that perform an action
  - Actions that manipulate data
  - Actions that analyze data
  - Actions that create graphics
- Advanced users can write their own commands and share them with others (.ado files)

Command Structure

- Commands, like verbs, have different syntax and only work properly in certain contexts:
  - Some can be used without objects (sleep or **`describe`**)
  - Some must have one of more objects (give or **`label`**)
  - Some really only work with objects as well as options (take umbrage or **`encode`**)

Command Basics

```
[by] command variable [if] [in], options
```

- This is the basic syntax of most Stata commands
- Stata commands have this built-in syntax to easily allow for the most common ways you might want to manipulate or analyze your data
- We will take a look at each one of these components one at a time

Command `[if]`

```
command variable if expression
```

- The if qualifier allows you to perform a command on a subset of your observations defined by the expression
- Adding an if is optional, not necessary. Without it, the command will be performed on all observations of a variable in the dataset
- If expressions are often used to define new variables or modifying existing variables, but could also be used to present analyses of subsets

Command `[if]`

- **An example of using if in a variable generation step:**

```
generate great_headroom = 1 if headroom > 3
```

```
replace great_headroom = 0 if headroom <= 3
```

- This defines a new variable `great_headroom` as 1 if the headroom variable is greater than 3 and 0 if the headroom variable is less than or equal to 3

- **An example of using if to perform an analysis on a subset of observations:**

```
summarize price if headroom > 3
```

- This performs and outputs the summarize command on only observations whose headroom variable is greater than 3

Command `[in]`

```
command variable in indices
```

- The in qualifier allows you to perform a command on a subset of observations based on their indices

- Adding an in is optional, and is most often used alongside a list command to take a look at certain low or high values:

```
list price weight in 1/5
```

- Remember that this indexing can change depending on how the observations are sorted — it is good practice to only use in after an explicit **`sort`** command

Command `by`

```
by variable_name: command ...
```

```
bysort variable_name: command ...
```

- The **by** prefix command allows you to perform stratified commands across values of a `variable_name`
- If the data is not sorted by the variable_name, an error will usually occur. To automatically sort, use the **`bysort`** command

```
bysort foreign: summarize mpg
```

- This will give summaries of the `mpg` variable stratified by the foreign `variable`
- This prefix works with continuous and categorical variables (but only really makes sense with categorical ones)

Command `options`

`command` `. . . , options`

- Almost all commands have options that allow the user to alter the performance of the command, display less or more detailed results of a command, or override regular Stata behavior

- Options are often unique to a command, but here are a couple common ones:

`command` `. ,` `replace` - overwrites the current file / variable

`command` `. ,` `clear` - clears away old data when loading or reading in files

`command` `. ,` `gen(`*`newvar`*`)` - uses the output of the command to create a new variable with name `newvar`

`command` `. ,` `detail` - prints more detailed output of a command

Explore Your `options`

- If you are wondering if you can do something in Stata, the best way to find out is through exploring the `options` in the **`help`** documentation for a command that is *close* to what you want to do

- Let's explore some basic statistical commands

summarize

```
summarize [variable_name]
```

- This command displays summary statistics for a variable (or all variables in a data set)

```
summarize variable_name, detail
```

- Provides more detailed summary statistics on a variable

```
bysort variable_name: summarize variable_name, detail
```

- Creates detailed stratified summary statistics for a variable

correlate

```
correlate variable_name1 variable_name2 ...
```

- This command calculates the correlation (or correlation matrix) between variables
- It needs at least two `variable_names` to work

```
pwcorr [variable_name1 variable_name2 ...]
```

- With no specifications, it will create a pairwise correlation matrix for whole dataset

```
pwcorr [variable_name1 variable_name2 ...], sig
```

- The sig option also calculates the significance of a correlation

tabulate

```
tabulate variable_name1 variable_name2
```

- This command will create a one-way or two-way table of values (depending on the number of `variable_names` given)

- A very commonly used command in epidemiology and a good first step to check on cell size for analysis

```
tabulate foreign great_headroom if price < 7000
```

- Using the if qualifier we can look at a table of a subset of observations

tabstat

```
tabstat variable_name1 variable_name2 ...
```

- This command creates a very customizable table of summary statistics for variables in a dataset

- Using `help tabstat` and clicking on the statistics options we can look through all the possible ways to build up a table

ameans

```
ameans variable_name1 ...
```

- This command creates a table of pythagorean means with confidence intervals

- Is there a difference between **ameans**, **gmeans**, **hmeans**? How can we check?

Exercises (1)

1. Titanic Data

   A. Open up your titanic.do file and run it, but change it so that it keeps passengers of all ages and survival statuses.

   B. Create a new categorical age variable over_30. Observations with an age over 30 should be assigned 1, those under 30 should be given a 0. Watch out for missing!

   C. Give your new variable and its values appropriate labels.

   D. Create a 2x2 table of over_30 variable and the survival value. Create a note for the over_30 variable which indicates how many people over 30 survived the titanic.

   E. In one command, have Stata find the mean ages of people who survived and people who did not survive. Add this information as a note to the survived variable.

   F. What is the sex variable's type? Create a new variable that can be used by Stata commands.

   G. Record the number of females who did not survive as a note in your new sex variable

Exercises (2)

1. Movie Metadata

    A. Open up your movies.do file and run it.

    B. Create a table that reports the mean, count, 25th percentile, 75 percentile, and range of all the continuous variables in the dataset. (remember **help**)

    C. Explore if any of the continuous variables are correlated, include their statistical significance.

    D. Re-create part B, but perform the command across categories of the expensive variable.

Saving Data and Merging Datasets

Saving Data

- Stata makes it easy to save data using the **`save`** command:

```
save filename
```

- Since Stata always has only one dataset in memory at a time, it is always clear what data we are saving

- Similar to the generate vs. replace commands for variables, Stata only lets you overwrite a file on disk if you use the replace option:

```
save filename, replace
```

Saving Files

- Always important to think about your project organization when saving .dta files
  - What is my current working directory? Use `pwd` command to check you are where you think you are
  - Should I be saving this file in a new folder, or under a new name?

Prepare for Merging

- We will be using some toy Stata datasets that are available via the **`webuse`** command
    - To start out, we'll load the `autosize` and `autoexpense` datasets

```
webuse autosize, clear

save autosize.dta
```

Overwriting Saved Files

- When overwriting a saved file with a standard `save` command we get a warning, so we use

```
save autosize.dta, replace
```

- Stata users often get so used to the `replace` option that they use it everywhere in their scrips

- This can lead to heartbreak!

Merging Datasets

- In Stata, combining the columns, or variables, of two or more datasets is called "merging" them (another common term for this process is "joining" datasets)
- In order to **`merge`** datasets, they must share at least one common variable (there must be a way to link them together)
- Stata refers to the dataset in memory as `master`, and to the additional dataset(s) as `using`
- The type of link is described using `master` : `using`, for example —
  - 1:1 (one observation in `master` is linked to one observation in `using`)
  - 1:many (one observation in `master` is linked to many observations in `using`)
  - many:1 (many observations in `master` are linked to one observation in `using`)
  - many:many (many observations in `master` are linked to many observations in `using`)

`merge 1:1`

- We will start out with a merge between datasets with a 1:1 variable link
- Let's create a new .do file and move through this process together:

```
merge 1:1 linking_variable using file_to_merge
```

```
merge 1:1 make using autoexpense
```

- This will result in a dataset loaded to memory with all the variables in `autosize` and all the variables in `autoexpense`

`merge 1:1`

- In the Stata Results window we will see a merge summary, showing the results of the merge
- Notice the `_merge` variable column:
  - A new variable `_merge` has been added to our dataset. This variable indicates the status of each observation (row) after the merge.

_merge = 1	The observation is present only in the master dataset
_merge = 2	The observation is present in one of the using datasets (but not the master datatset).
_merge = 3	The observation is present in at least two datasets, either master or using

`merge 1:1`

- In this particular example we see that there is one observation from the `master` dataset that is not matched in the `using` dataset (`Plym. Arrow`)
- We see that the values from the `price` and `mpg` variables from the `autoexpense` dataset are left blank (missing) for this observation

merge, assert

- We can use the assert option with an argument to automatically check on the status of our merge

```
merge 1:1 linking_variable using
file_to_merge, assert(match)
```

```
merge 1:1 linking_variable using
file_to_merge, assert(match master)
```

`merge 1:m`

- The syntax for these merges is largely the same, except for `1:m` replacing `1:1`

```
merge 1:m linking_variable using
file_to_merge
```

- The Stata toy datasets `overlap2` and `overlap1` can be used for this purpose

`merge` options

- To keep only observations with that have matched, use the option `merge, keep(match)`
- To perform a merge without producing the _merge variable, use the option `merge, nogen`

Exercises (1)

1. Auto Data

    A. Save your auto data as auto.dta.

    B. Now split your auto data into two files: save one as *auto_cont.dta* that contains **make** and all **continuous variables**, save the other as *auto_other.dta* that contains **make** and all the **non-continuous variables**.

    C. Open auto_cont and perform a 1:1 merge using auto_other. Save this dataset as auto_merged. How can you make it identical to your original auto.dta file? Do it and save.

    D. Load your original auto.dta file. Replace the values of the headroom variable with the weight variable.

    E. Perform a 1:1 merge with the auto data you currently have in memory with the auto.dta file saved on disk. What happened to headroom? What does this tell you about merging datasets who share variable names?

Exercises (2)

1. m:1 Merging Example

   A. Create a do-file named merging.do

   B. Copy the following series of commands into your do-file. Use comments in your file to explain step-by-step what is happening. Don't be afraid to explore **help merge!**

```
Setup
    . webuse overlap1, clear
    . list, sepby(id)
    . webuse overlap2
    . list

Perform m:1 match merge, illustrating update option
    . webuse overlap1
    . merge m:1 id using http://www.stata-press.com/data/r14/overlap2, update
    . list
```

```
Perform m:1 match merge, illustrating update replace option
    . webuse overlap1, clear
    . merge m:1 id using http://www.stata-press.com/data/r14/overlap2, update replace
    . list
```

Exploring Data in Stata Using Plots

Graphing Basics

- Visually inspecting data is always an important first step to data analysis
- Stata allows users to quickly and easily create scatter, line, and bar graphs to explore their datasets
- The main command for graphing is graph, followed by a keyword that identifies the type of graph:

```
graph twoway scatter vary varx

graph box cont_var, over(cat_var)

graph twoway line vary varx

graph bar (mean) cont_var, over(cat_var)
```

Graphing Window

- When the user runs a **`graph`** command, the Graph Window will automatically pop up

- Stata has a built-in graph editor that is fairly customizable. It can be useful for touching up a graph for publication, but nearly all of the options available there are also available through **`graph`** command `options`

- If you close the Graph Window, you can always return to the last created graph by typing a blank **`graph`** command into the Command Window

graph twoway scatter

```
graph twoway scatter vary varx, title("Title Text")
```

- This command creates a scatterplot with *vary* on the y-axis and *varx* on the x-axis
- The title option adds "Title Text" as a title
- There are many other options, such as `msymbol`, `mcolor`, or `msize` which control the type of plot symbol, the color of symbols, and their size
- Let's try graphing `mpg` vs. `price` of automobiles

graph box

```
graph box cont_var, over(cat_var) title("Text")
```

- This command creates box plots of a continuous variable (`cont_var`) separated by a categorical variable (`cat_var`)
- Stata will allow you to create group box plots over continuous variables
- We can take a look at `mpg` across domestic and foreign cars as an example
- There are many other options, which are best explored by calling **`help`** graph box

graph twoway line

```
graph twoway line vary varx, lpattern(dash)
```

- This command creates a line graph (with a dashed line)
- Data need to be set up properly, but any number of y-variables can be plotted on the same scale by repeating
- We can take a look at `le` over `time` using the `uslifeexp` dataset
- Twoway graph types can be easily layered using the || device:

```
graph twoway line vary varx || twoway scatter vary varx
```

graph bar

```
graph bar (mean) cont_var, over(cat_var)
```

- This command creates a bar graph of a continuous variable over categorical groups using the `mean` statistical function
- The user can specify a different statistical function or use `asis` to graph the value of a variable
- The cont_var is optional, without it Stata produces a bar graph of percentage of observations in each cat_var

```
graph bar, over(foreign)
```

graph name

```
sysuse educ99gdp

graph bar (asis) public, over(country) name(public_graph)

graph bar (asis) public private, over(country) name(comparison_graph)
```

- The name option will save your graph to Stata's working memory (not as a file someplace on your hard drive)
- To re-display a previously named graph, use **display** `graph graph_name`

graph combine

```
graph combine graph1 graph2 ...
```

```
graph combine public_graph comparison_graph
```

- The combine command takes multiple graph names and combines them into one graph
- There are several options for arranging your combined graphs, notably `rows(n)` and `columns(n)` which determine how many rows and/or columns of graphs the combined graph will have

graph save

```
graph save [graphname] filename
```

- The graph save command will save the current graph or the graph `graphname` as a file to disk
- Stata graphs are saved as .gph files, which are Stata-specific file types. When loaded, these files will look different based on individual user settings
- If you want a graph to look *exactly* the way it does on your screen, you should save using the option `asis`

graph export

```
graph export filename.suffix
```

- This command exports the current graph as a specific image file (using the appropriate file extension `.suffix`)

- Each file type has a specific set of options (often tied to size of the saved image and their resolution)

```
graph display comparison_graph
```

```
graph export comparison_graph.tif
```

Exercises (1)

1. Movie Metadata

   A. Create a scatter plot of gross against budget. ***It doesn't look great!*** Scour the graph help files to find out how to use the log scale for both x and y axes, and name this graph gross_budget_log.

   B. Create a box plot which compares the imdb scores of movies across the expensive categorical variable. Name it imdb_expensive.

   C. Create a bar graph that displays the percentage of movies over the categorical country code variable. Name it movies_country.

   D. Combine the three graphs above into one graph. Make one version with one column, make another with one row. Save these graphs as movies_col.gph and movies_row.gph!

Exercises (2)

1. Auto Data

   A. Write a loop in auto.do which produces separate scatter plots for price vs mileage, price vs headroom, and price vs trunk space. Make sure each one has a title! Save each graph as price_*othervariable*.gph.

   B. Create box plots of mileage and headroom across our price category variable. To make things more complex, use the by option to create separate graphs across the foreign variable. Give them names.

   C. Combine the graphs from part B and export this combined graph as a .tif file named auto_boxplots.tif