

Using Stata for Data Work

Workshop Page: <http://www3.nd.edu/~jng2/workshop>

Agenda

0. Preliminaries
1. Exercise 1
Loading fixed format ASCII data
2. Exercise 2
Generate scatter plot of vehicle gas mileage over price
 - a. Cleaning data: handling string variables
 - b. Graphing: scatter plot
3. Exercise 3
Reproduce table and graph of happiness and GDP over time
 - a. Merging two datasets
 - b. Generating new variables and labeling them
 - c. “Collapsing” observations
 - d. Graphing: two line plots on one graph, each with its own y-axis

BREAK

4. Exercise 4
More data cleaning and manipulation
 - a. Using Stata’s indexing features
 - b. The egen suite of commands
 - c. Reshaping data
 - d. Dealing with dates
5. Exercise 5
Produce publication-quality tables of summary statistics and regression results
6. Exercise 6
Using for-loops for repetitive tasks
7. Exercise 7
Match husbands and wives from NHIS 2012 data
8. Exercise 8
Produce a map of unhappiness by Census division

0. PRELIMINARIES

Stata is a statistical software package widely used by quantitative social scientists (e.g. economists, sociologists, political scientists). One of the most attractive things about Stata is its extensive collection of commands that can be used to easily accomplish virtually any manipulation and analysis of data that a researcher would need. This is no accident: Stata was written by a bunch of economists in 1985, so it is geared towards the needs of social scientists.

A. Layout of Data

You can think of data loaded into Stata as a spreadsheet, with each row containing an observation, and each column containing a variable.

B. Variable Names

Variables are case sensitive. For example, a variable named AGE is distinct from a variable named age. Variable names cannot start with a number and can contain only alphanumeric characters. Exception to the rule: the underscore is allowed in variable names.

C. Sending Commands


You instruct Stata to accomplish specific tasks by sending specific commands. There are three ways to send commands in Stata:

1. Point and click using the drop down menus.
2. Enter commands in the command line interface.
3. Create a “do-file” containing commands, which you will then execute.

D. Do-Files

Anytime you expect to work on a project in more than one sitting, you should use a do-file. Why should you work with do-files?

A do-file contains every command that you ever used for your project, from the very first step (loading data) to the very last (exporting your results). It documents every step you took in the process of manipulating and analyzing data. If you need to modify or repeat certain steps, you simply modify your do-file appropriately instead of redoing everything.

To create a do-file, in Stata click on the icon on the toolbar that looks like this:  (Note: If you hover over the icon, a label saying “New Do-file Editor” appears.)

Alternatively, on the top menu bar, choose Window > Do-file Editor > New Do-file Editor.

E. Help

Stata has a very extensive built-in help system. To pull up the help file for a particular command, say describe, simply enter `help describe` into the command line interface. Googling also usually yields useful results.

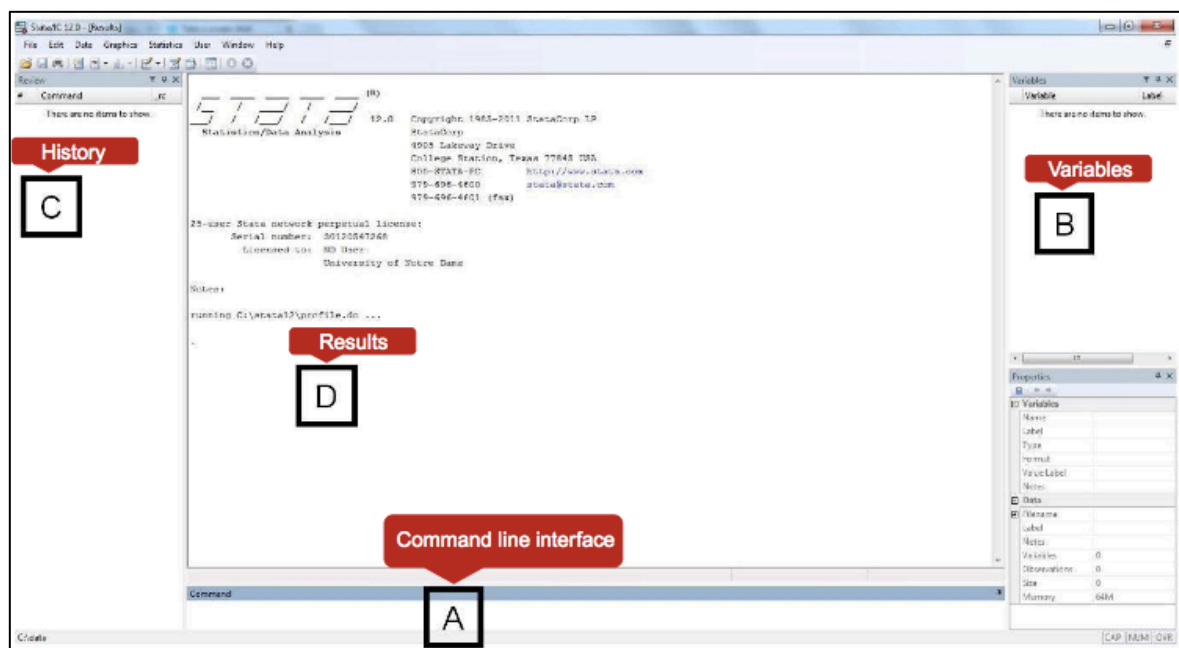
F. The Stata Environment

Box A: This is the command line interface. This is where you type in your commands, similar to what you would do on any Unix-type terminal.

Box B: This is where all the variables in the dataset that is currently in memory are displayed.

Box C: The “review window” is where a history of all commands you have entered is displayed. This history vanishes when you terminate Stata.

Box D: This is where the result of each command entered is displayed.



G. Operators

Assignment operator: =

Relational operators:

equals to: ==

not equals to: != (alternatively ~=)

less than: <

less than or equal to: <=

greater than: >

greater than or equal to: >=

Logical operators:

AND: &

OR: |

NOT: ! (alternatively ~)

Conditional statements:

The `if` conditional statement can be used two ways:

1. As a qualifier at the end of a command—`if` at the end of a command means the command is to use only the data specified. The `if` qualifier is allowed with most Stata commands. For example, the following command regresses `gdp` on `happy` using observations between the years 1975 and 1997.


```
regress gdp happy if year>=1975 & year<=1997
```
2. As a programming command, like you would in any programming language. The syntax for `if` is strict; for illustrative examples, see
 - a. Exercise 6, Example 4
 - b. Stata's help file for the `if` command (`help ifcmd`).

1. EXERCISE 1

We will start by loading data from the 2012 wave of the National Health Interview Survey (NHIS).

- Go here: http://www.cdc.gov/nchs/nhis/nhis_2012_data_release.htm and expand the line that says “Data Files”.
- Download and extract the household file (“ASCII data”) onto your computer.
- The file you just extracted, **househld.dat**, is an ASCII file in fixed format.
- NHIS helpfully provides a Stata script (do-file) that loads the data into Stata. Download and save “Sample Stata statements” – this is the file called **househld.do**. We use this do-file to load the household data into Stata.
- Be sure both **househld.dat** and **househld.do** are stored in the same folder on your computer.
- Open **househld.do** in Stata. Add the following three lines to the start of the do-file.

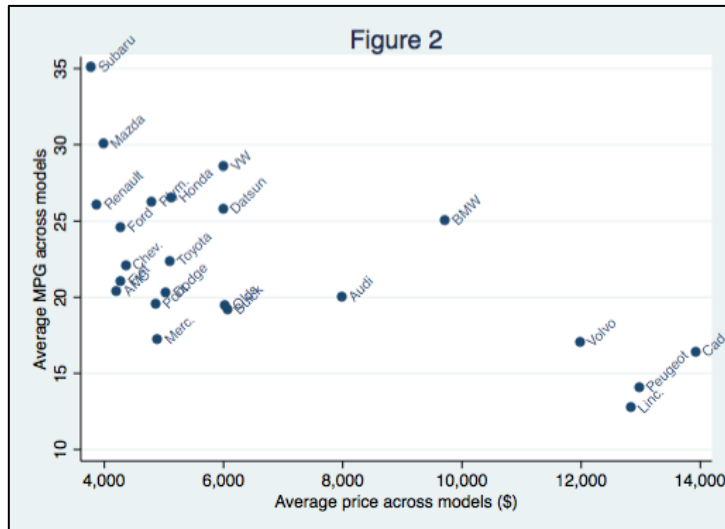
```
cd Your_Folder
set more off
capture log close
```

where **Your_Folder** is the full path to the folder containing **househld.dat**.

- The last two commands above modify default Stata behavior for your convenience. `set more off` forces the do-file to execute without pausing as the results screen fills up. `capture log close` forces Stata to ignore a log file that was not explicitly closed. If the purpose of these commands is unclear, it will become clear in a second when we run the do-file.
- Execute the do-file. This will load the data contained in **househld.dat** into Stata (using the `infix` command), format it nicely with variable and value labels, and save it as a Stata-format file in **Your_Folder**.

2. EXERCISE 2

The goal of this exercise is to produce **Figure 2** below.



STEP 1

- Load the following dataset into Stata using the `sysuse` command.¹

```
sysuse auto, clear
```

STEP 2

- Using the string function called `word`, extract automakers from the variable `make`, saving them in a new variable called `automaker`.²

```
gen automaker = word(make,1)
```

STEP 3

- Next, collapse the data to obtain the average `price` and `mpg` across models for each `automaker`.

```
collapse (mean) avg_price = price avg_mpg = mpg, by(automaker)
```

STEP 4

- Produce the graph using the `twoway scatter` command. You can break up a long command using three slashes, as the code below shows.

```
twoway scatter avg_mpg avg_price, ///  
mlabel(automaker) mlabangle(45) ///  
ytitle("Average MPG across models") ///  
xtitle("Average price across models ($)")
```

¹ The `sysuse` command loads into memory an example dataset from Stata's default directory on your computer's hard drive. It is very useful for playing around with commands. To see a list of all available example datasets, type `sysuse dir`.

² For a list of string functions, type `help string`.

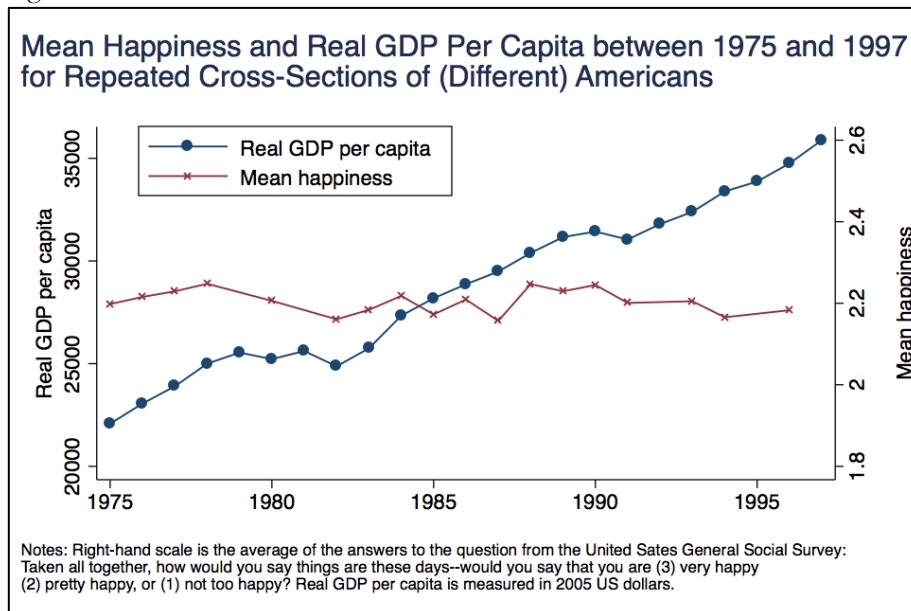
3. EXERCISE 3

The goal of this exercise is to produce **Table 1** and **Figure 1** below.

Table 1: Annual real GDP per capita and mean happiness, 1975-1997

year	rgdppc	meanhappy
1975	22075.38	2.19798
1976	23055.56	2.215477
1977	23896.55	2.229208
1978	24999.11	2.247858
1979	25538.14	
1980	25220.93	2.205882
1981	25615.52	
1982	24869.67	2.160647
1983	25745.56	2.183725
1984	27333.99	2.217993
1985	28183.51	2.172549
1986	28864.97	2.20911
1987	29485.49	2.156742
1988	30390.34	2.24693
1989	31163.08	2.229358
1990	31430.6	2.244673
1991	31046.02	2.200798
1992	31796.23	
1993	32392.62	2.204872
1994	33384.2	2.165603
1995	33868.18	
1996	34749.89	2.183016
1997	35882.78	

Figure 1



Source: <http://www.jstor.org/stable/30033632?origin=JSTOR-pdf>

STEP 1

- Save **gss_happy.dta** and **pwt_gdp.dta** onto `Your_Folder` from the Workshop Page -- <http://www3.nd.edu/~jng2/workshop> -- where `Your_Folder` is the full path to a folder of your choice on your computer.

STEP 2

- Start a new do-file and save it in `Your_Folder`.
- On the very first line of the do-file, type the following:

```
cd Your_Folder
clear all
set more off
```

If `Your_Folder` contains whitespace, you will have to enclose the entire path in double quotes.

- The `cd` command stands for “change directory”; it instructs Stata to treat `Your_Folder` as the current working directory. Unless you specify otherwise, once the working directory is set, Stata will look in the working directory for data files to read and will save generated data files in the working directory.
- To find out your current working directory, use the `pwd` command.
- To get Stata to execute a single line or successive lines of code from a do-file, highlight the line/s and click the “Do” button in the menu bar of the do-file.

STEP 3

- Load **gss_happy.dta** into Stata using the `use` command.³ So that you do not lose your work, enter the command in the do-file, and then execute it.

```
use gss_happy, clear
```

STEP 4

- To see a list of all the variables, variable types, and sample size, type `describe`.
- To obtain summary statistics of all the variables in the dataset, type `summarize`.
- To browse the raw data in spreadsheet form, type `browse`.
- To obtain frequency counts of the values of the variable **happy**, type `tab happy`. Compare this to

```
tab happy, nolabel
```

- Try this: what happens when you enter `tab happy, missing`?

STEP 5

- Notice that the values for **happy** are the opposite of what’s been coded in Figure 1. We need to recode this variable. We can do so by generating a new variable called **happynew** that takes on the values that match what we see in Figure 1.

```
gen happynew = happy
replace happynew = 3 if happy == 1
replace happynew = 1 if happy == 3
```

³ Stata commands to load ASCII files are `infix` and `infile`. Stata can read Excel spreadsheets with `import excel`.

- To confirm that **happynew** was coded correctly, cross-tabulate **happy** and **happynew**:

```
tab happy happynew, nolabel
```

STEP 6

- Data for the happiness variable are at the person level, but Table 1 and Figure 1 show annual average happiness across all persons at the country level.
- To obtain average happiness across all persons for each year, use the `collapse` command, like so:

```
collapse (mean) meanhappy=happynew, by(year)
```

- This collapses the data and generates a variable, **meanhappy**, which contains annual average happiness at the country level.
- You can label the variable to make it more descriptive for yourself and others.

```
label var meanhappy "Mean happiness: 3-very, 2-pretty, 1-not too"
```

STEP 7

- We only need observations from 1975 through 1997, so we will keep only observations from that range of years.

```
keep if year>=1975 & year<=1997
```

- Finally, save the current data in a new file using the `save` command as below. This file will be saved in your current working directory. The file will be in Stata format and have the extension `.dta` (you do not have to specify the extension).

```
save colgsshappy, replace
```

- Note: If want to avoid creating too many intermediate files, you can choose to save data as a temporary file. A temporary file exists only in RAM and is not saved to your hard drive. To save data as a temporary file, you first specify a name for the temporary file, and then save it as a local macro.

```
tempfile tempgsshappy  
save `tempgsshappy', replace
```

When declaring a macro for the first time, no punctuation is needed. However, to use an existing macro, strict punctuation rules must be followed. Local macros are punctuated by an opening backtick (```) and closing right single quote (`'`). The backtick is found on the same key as the tilde (`~`) on the upper right corner of your keyboard.

STEP 8

- Stata only handles one dataset at a time. To use variables stored in two separate datasets, you must combine the datasets.
- Remember that our end goal in this exercise is to combine the happiness and GDP datasets.
- We merge two datasets across observations using the `merge` command. Think of it as adding new columns to an existing spreadsheet.⁴

⁴ If you need to add new observations to existing data, the command for that is `append`. Think of this as adding new rows to an existing spreadsheet.

- To merge two datasets, you need a variable (or set of variables) that is common to the two datasets. This variable is the “identifier”; it identifies each observation (row).
- Then, you will need to figure out whether each observation of the identifier variable appears only once, or whether it repeats. You do so using the `duplicates report` command.
- In the present context, our goal is to merge annual GDP data to annual happiness data. Therefore the identifier variable is **year**.
- In the happiness dataset that we have open, it is obvious that **year** uniquely identifies each observation when we browse the raw data.
- To confirm that the observations are uniquely identified by the **year** variable, type

```
duplicates report year
```

- This returns the following result, which tells us that 18 observations (rows) in the data have unique **year** values, and that there isn’t a year value that repeats over multiple observations. Conclusion: in **colgsshappy.dta**, **year** uniquely identifies each observation.

```
-----
      copies | observations      surplus
-----+-----
           1 |             18           0
-----
```

STEP 9

- Stata only handles one dataset at a time. Now that we have saved the data containing happiness, we will work with GDP data, and then merge the GDP data with the happiness data. Open **pwt_gdp.dta**.

```
use pwt_gdp, clear

keep if year>=1975 & year<=1997

gen rgdppc = rgdpna/pop
label var rgdppc "Real GDP per capita"
```

STEP 10

- Verify that the observations are uniquely identified by **year** variable (see STEP 8).

```
duplicates report year
```

STEP 11

- We are now in a position to merge the GDP data to the happiness data stored in **colgsshappy.dta**.

```
merge 1:1 year using colgsshappy
```

- Stata refers to the dataset that you have open as the “master dataset”. It refers to the dataset that you add to the master dataset in a merge operation as the “using dataset”.
- Here, the master dataset is the GDP dataset that we currently have open, and the using dataset is **colgsshappy.dta**.
- We just performed a “one-to-one” merge. We knew that a one-to-one merge was what was needed because the identifier variable used to merge the two datasets, **year**, uniquely identifies observations in both the master and using datasets.

- If the situation calls for it, you would perform a “one-to-many” (1:m) or “many-to-one” (m:1) merge.⁵

STEP 12

- If you browse the resulting data, you will find that it contains the information in Table 1.
- To sort the data in ascending order of year, type

```
sort year
```

- To output the data to an Excel spreadsheet, use the `outsheet` command.

```
outsheet year rgdppc meanhappy using table1.xls, replace
```

STEP 13

- The command to plot a graph is `twoway` followed by a subcommand that specifies the type of graph.
- To plot a connected line graph of GDP over time, type

```
twoway connected rgdppc year
```

- To plot happiness over time, type

```
twoway connected meanhappy year
```

- We need these two plots in the same graph. To do so, the basic command is (in one line):

```
twoway ( connected rgdppc year, yaxis(1) ) ( connected meanhappy  
year, yaxis(2) )
```

- Alternatively,

```
twoway connected rgdppc year, yaxis(1) || connected meanhappy  
year, yaxis(2)
```

- To save the graph in Stata’s `.gph` file format, type

```
graph save happygdp, replace
```

- To export the graph to an external format such as PNG or PDF, type

```
graph export happygdp.png, replace
```

- Complete commands are given in the do-file, **happygdp.do**, which you can download from the Workshop Page.

⁵ Stata also allows a many-to-many merge, but in many scenarios that I have encountered, many-to-many merges should be avoided. If you think a many-to-many merge is needed, you probably need to work on the data further or use `joinby`.

4. EXERCISE 4

Some more ways to clean and manipulate data:

- A. Using Stata's indexing features
- B. The egen suite of commands
- C. Reshaping data
- D. Dealing with dates

A. Using Stata's indexing features

Stata has two system variables that always exist as long as data is loaded, `_n` and `_N`. `_n` indexes observations (rows): `_n = 1` is the first row, `_n = 2` is the second, and so on. `_N` denotes the total number of rows.

To illustrate, let's use **stocks.dta**. It contains longitudinal data for a fictional stock portfolio.

STEP 1

```
use "http://www3.nd.edu/~jng2/workshop/stocks.dta", clear
browse
```

The data aren't sorted in any particular order. Before we do anything else, it makes sense to first sort the data:

```
sort stockid year
```

This sorts observations first by stockid then by year, in ascending order.

To get a sense of what `_n` and `_N` can be used for, enter these two commands:

```
gen obsnum = _n
gen totnum = _N
```

STEP 2

`_n` and `_N` can also be used within subgroups. To generate counters like the above for each stockid, do this:

```
use "http://www3.nd.edu/~jng2/workshop/stocks.dta", clear
sort stockid year
by stockid: gen obsnum = _n
by stockid: gen totnum = _N
```

Equivalently, instead of sorting unsorted data prior to `by`, use `bysort`:

```
bysort stockid (year): gen obsnum = _n
bysort stockid (year): gen totnum = _N
```

STEP 3

Now that you have a sense of what `_n` and `_N` do, let's use `_n` in combination with `by` to perform a concrete task. We will fill in the blanks in the ticker variable (this assumes that the ticker symbols for these stocks did not change over time).

First, make sure that the data are sorted by stockid and year.

```
use "http://www3.nd.edu/~jng2/workshop/stocks.dta", clear
sort stockid year
```

To fill in row 6 of ticker with the appropriate value ("AMZN"), you could do this:

```
replace ticker = "AMZN" in 6
```

To fill in rows 6 through 10 with "AMZN", you could do this:

```
replace ticker = "AMZN" in 6/10
```

Clearly, with multiple observations to replace, replacing values line-by-line this way becomes cumbersome and prone to human error. We can automate the process using `by` and `_n`.

```
bysort stockid (year): replace ticker=ticker[1] if _n>1
```

The above command states that within each `stockid` group (that has been sorted by ascending order of year), all ticker observations except the first are to be replaced with the value from the first year.

STEP 4

Notice that we messed up since the tickers for `stockid==5` and `==6` are now all blanks. We need to modify our algorithm to avoid overwriting ticker values with blanks. Reload the original data and start from scratch.

```
use "http://www3.nd.edu/~jng2/workshop/stocks.dta", clear
```

The years don't really matter for this task, so instead of sorting by `stockid` and year, let's sort in ascending order of `stockid` and then descending order of `ticker`:

```
gsort + stockid - ticker
```

As you can see, the first observation within each `stockid` now contains the correct ticker symbol, so filling in the blanks is now a simple matter of

```
by stockid: replace ticker=ticker[1] if _n>1
```

and we're done.

B. The egen suite of commands

The `egen` command consists of functions that extend the capability of the `generate` command. The various functions within `egen` create variables that hold information about patterns and calculations within subgroups or across columns.

A couple of examples:

Example 1

The dataset **stock.dta** has longitudinal data on (fictional) annual returns for a bunch of stocks. Find the highest return for each stock.

```
use "http://www3.nd.edu/~jng2/workshop/stocks.dta", clear
bysort stockid: egen maxreturn = max(return)
```

This creates a new variable `maxreturn` that holds the highest value of return across all observations of each stockid. For each stockid, find the year/s that yielded the highest return.

```
list stockid year if return == maxreturn
```

Example 2

Count the number of observations for each stockid.

```
bysort stockid: egen numobs = count(stockid)
```

(Note: You can accomplish the same thing with `tabulate stockid` or `duplicates report stockid`.)

C. Reshaping data

Sometimes you'll need to reshape data from long to wide or vice versa. An illustration:

Reshape data from long to wide:

```
reshape wide city sex, i(person) j(year) reshapes "long" data to "wide".
```

Reshape data from wide to long:

```
reshape long city sex, i(person) j(year) reshapes wide data to long.
```

Long data				Wide data				
person	year	city	sex	person	city2010	sex2010	city2011	sex2011
A	2010	Los Angeles	male	A	Los Angeles	male	San Francisco	male
A	2011	San Francisco	male	B	Chicago	female	Chicago	female
B	2010	Chicago	female	C	Indianapolis	male	Los Angeles	male
B	2011	Chicago	female	D	Detroit	female	Chicago	female
C	2010	Indianapolis	male					
C	2011	Los Angeles	male					
D	2010	Detroit	female					
D	2011	Chicago	female					

Let's reshape the Stata example dataset `bplong.dta` from long to wide.

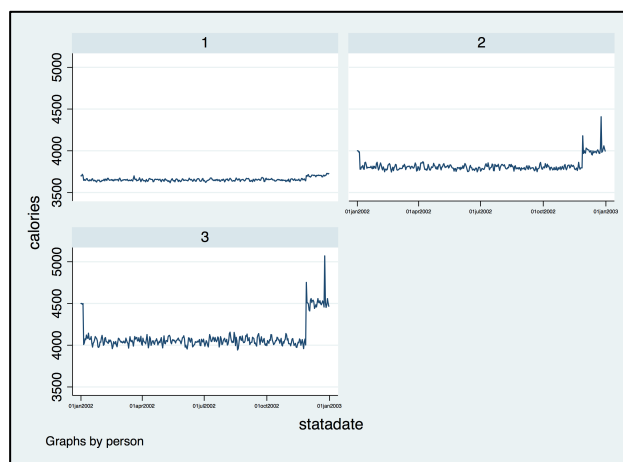
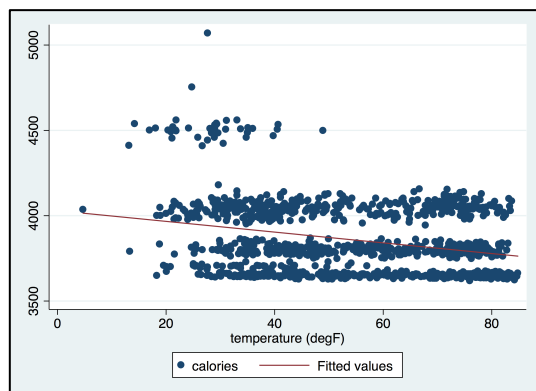
```
sysuse bplong, clear
reshape wide sex agegrp bp, i(patient) j(when)
```

One would usually use long form data for time series or panel data analysis and wide form for cross sectional.

D. Dealing with dates

Scenario: Suppose you want to investigate whether daily calorie intake is correlated with temperature (you suspect people eat more when it's cold outside). You have a (fictional) dataset on daily calorie intake for a sample of Indiana residents. This dataset also identifies the weather station closest to each person. In order to answer your research question, you will need to merge this dataset to another dataset on daily temperatures measured at the weather stations.

Our end products in this exercise will be these two graphs.



STEP 1

Save the calorie intake data, **dailycalories.dta** from the Workshop Page to your working folder and open it.

The date variable, `day`, is a string variable. There are a few ways to confirm that it's a string:

1. Type `describe day`. The results window tells us that `day` is of type `str9` (a string that's 9 characters in length).
2. Click on `day` in the "Variables" window. The "Properties" window gives us the same information as the above.
3. Browse the raw data (type `browse`). Strings appear in red.

The correct way to handle dates in Stata is to convert them to a number measured in days elapsed since January 1, 1960.⁶ For dates stored as strings, the `date()` function does the conversion flawlessly. Here's how to do it:

```
gen statadate = date(day, "DMY")
label var statadate "date in days elapsed since 1-Jan-1960"
```

This creates a variable called `statadate` that stores the dates in terms of days elapsed since January 1, 1960. If you browse the data now you'll see that `statadate` is simply a number. You could get Stata to display it as a date by entering `format %td statadate`.

Later, we will merge this dataset to temperature data. Each weather station measures the temperature each day, so the merging variables should be `stn` and `statadate`.

It just so happens that each person in this dataset lived in a different location, so each person has a different weather station. This means that each observation identified by the combination of `stn` and `statadate` should be unique, which we can verify using

```
duplicates report stn statadate
```

Save the calories data on your hard drive. We will merge it to daily temperature data in Step 3.

⁶ You do not *always* have to perform the conversion of dates to Stata's format. In this example, conversion is necessary because the dates are given to us as strings. However, if your dates are recorded as e.g. 01/31/2012, no adjustment is necessary--Stata will automatically interpret them as dates and store them in the appropriate format. In that case, when you `describe` the variable, you would see that the date variable is displayed in `%td` format.

```
save caloriesdata
```

STEP 2

Now open the daily temperature data, **dailytemp.dta**. This data came from [NOAA](#), specifically [GSOD](#).

Dates in this file are stored by the variable `yearmoda`, which is a number where the first four digits contain the year, the next two contain the month, and the last two contain the day. We want to convert it to days elapsed since January 1, 1960, but to do so we first need to split it into its constituent components. An easy way to split it is to use the add-on command `nsplit`.

```
nsplit yearmoda, digits(4 2 2) generate(yyyy mm dd)
```

We can then generate `statadate` to store dates in days elapsed since January 1, 1960, using the `mdy()` function:

```
gen statadate = mdy(mm, dd, yyyy)
```

Each station-date combination should be unique, which we verify using

```
duplicates report stn statadate
```

STEP 3

Picking up from where we left off in Step 2, remember that the dataset that's currently open is the daily temperature data. Now let's merge the calories data to it.

We previously determined in Step 1 that the combination of `stn` and `statadate` uniquely identifies each observation in the calories data. Therefore we must perform a one-to-one merge.

```
merge 1:1 stn statadate using caloriesdata
```

We can now explore the relationship between calorie intake and temperature.

A simple correlation coefficient suggests that this relationship is negative:

```
corr calories temp
```

We can visualize this relationship through a scatter plot:

```
twoway (scatter calories temp) (lfit calories temp)
graph export caltempscatter.png
```

Let's see how each person's calorie intake changed over the course of the year:

```
format %td statadate
twoway line calories statadate, xlabel(,labsize(tiny)) by(person)
graph export caldaybyperson.png
```

5. EXERCISE 5

The goal of this exercise is to produce publication-quality tables of summary statistics and regression results. We will first produce **Table 2** followed by **Table 3**. Refer to the do-file titled **nicetables.do** found in the Workshop Page.

Age	39.15 (3.060)
Hourly wage	7.77 (5.756)
Race:	
White	0.73 (0.445)
Black	0.26 (0.438)
Other	0.01 (0.107)
College graduate	0.24 (0.425)
<i>N</i>	2246

Standard deviations in parentheses.

	(1)
Dependent variable: Wage	
Age	-0.0738 (0.0382)
College graduate	3.525 (0.276)
Race:	
Black	-0.991 (0.268)
Other	0.155 (1.094)
Constant	10.08 (1.510)
<i>N</i>	2246
<i>R</i> ²	0.078

Standard errors in parentheses
The omitted race category is white.

These tables were produced using the `estout` suite of commands. It is an add-on; you can install it by simply typing `ssc install estout`.

6. EXERCISE 6

The goal of this exercise is to introduce you to the use of for-loops. Generally speaking, you can perform almost every manipulation imaginable using commands, without having to code up a loop.⁷ That said, loops are still very useful for performing repetitive tasks. For a simple example, see Example 1 below.

Loops in Stata adhere to the same general principles as in other programming languages. The following are the three types of loops in Stata. The use of each is best demonstrated with simple examples, using the built-in example dataset `auto.dta`, which you can load it using `sysuse auto`.

- `foreach`
- `forvalues`
- `while`

Example 1: `foreach`

```
*Objective: Attach the prefix _78 to the variables price and mpg
foreach v in price mpg {
    rename `v' `v'_78
}

*The following is equivalent:
foreach v of varlist price mpg {
    rename `v' `v'_78
}
```

Example 2: `forvalues`

```
*Objective: Count the number of observations in this dataset
local counter = 0
local N = _N

forvalues i = 1 / `N' {
    local counter = `counter'+1
}

display `counter'
*Note that this exactly what the count command does.
```

Example 3: `while`

```
*Objective: Count the number of observations in this dataset
local counter = 1
local N = _N

while `counter' < `N' {
    local counter = `counter'+1
}

display `counter'
```

⁷The `egen` suite of commands is particularly useful. Check it out.

Loops can be nested. For example:

Example 4: Nested loops

```
*Objective: Count the number of foreign and domestic cars
sysuse auto, clear
local count1 = 0
local count0 = 0
local N = _N

forval orig = 0/1 { forval
    row = 1/`N'{
        if foreign[`row'] == `orig' {
            local count`orig' = `count`orig'' + 1
        }
    }
}

display "No. of foreign cars:`count1'"
display "No. of domestic cars:`count0'"
*You can obtain the same information using the following command:
bysort foreign: count.
```

Loops are always used with **macros**. A macro has a “macro name” and “macro contents”. Everywhere a punctuated macro name appears in a command—punctuation is defined below—the macro contents are substituted for the macro name.

The contents of global macros are defined with the global command and those of local macros with the local command. See the above examples.

Global macros, once defined, are available everywhere until you quit Stata.

Local macros exist solely within the process in which they were first defined. A process may be an interactive Stata session, a do-file, a loop, or a program.

To see all the existing macros in your current Stata session, type `macro dir`.

Local or global macro -- which to use?

- Local macros are preferable in most situations.
- In loops, definitely use local macros, not global!
- Only create a global macro if you want to use it in different processes (say, across multiple do-files).
- A situation in which a global macro might be useful is if you want a list of variable names to be available to multiple do-files, to be used as control variables in various regression specifications, for example.

Punctuation for local and global macros

- To use a global macro, the macro name is punctuated with a dollar sign (\$) to the left.
- To use a local macro, the macro name is punctuated with a back tick (` --look for the tilde key if you can't find it) to the left and right single quote (') to the right.
- The examples above demonstrate how to define a macro and retrieve its contents.

7. EXERCISE 7

The objective of this exercise is to create a dataset of matched husbands and wives from NHIS 2012 data. You will make use of what you have learned today, plus some additional tips and tricks, to accomplish this task.

NHIS data is split into three separate files: households, families and persons. The household is the top-level unit. A household can have multiple families. Each family contains persons.

Link to NHIS 2012 data: http://www.cdc.gov/nchs/nhis/nhis_2012_data_release.htm

Your first task is to load and save the household, family and person files in Stata, keeping just the following variables:

File	Variables to keep
personsx.dta	hhx fmx fpx region sex r_maritl fspous2 phstat notcov
familyxx.dta	hhx fmx
househld.dta	hhx

Next, merge households, families and persons such that data at the person level is obtained. Save the resulting data as **nhis2012_select.dta**.

Hint: Households are uniquely identified by hhx. Families are uniquely identified by hhx and fmx. Persons are uniquely identified by hhx, fmx and fpx.

Question: After merging, how many households did not have any families interviewed in 2012? How would you drop these observations?

Use **nhis2012_select.dta**. First, answer these questions: Which variable tells you about an individual's marital status? Which variable tells you the person number of an individual's spouse? Are there individuals who are "married with spouse in household" but do not have a spouse person number?

From this data, generate couple-level data containing information about husbands and wives. Each **couple** should occupy one row. Your summary statistics of the resulting dataset should look like this:

Variable	Obs	Mean	Std. Dev.	Min	Max
hhx	0				
fmx	0				
fpx_f	0				
region_f	20,982	2.752693	1.036159	1	4
sex_f	20,982	2	0	2	2
r_maritl_f	20,982	1	0	1	1
fspous2	0				
phstat_f	20,982	2.261367	1.074346	1	9
notcov_f	20,982	1.895005	.6373836	1	9
male_f	20,982	0	0	0	0
wifeid	0				
fpx_m	0				
region_m	20,982	2.752693	1.036159	1	4
sex_m	20,982	1	0	1	1
r_maritl_m	20,982	1	0	1	1
phstat_m	20,982	2.279716	1.074183	1	9
notcov_m	20,982	1.889572	.6173824	1	9
male_m	20,982	1	0	1	1

Question: Are there families with more than one couple? How would you identify and tag them?

Question: How would you create a unique identifier for each couple?

Question: What percentage of couples have at least one spouse covered by health insurance?

Question: What is the correlation between health of husband and health of wife?

Reshape the couples dataset to long form, such that couple information occupy two rows: one for the husband and one for the wife. Then, reproduce the table below.

Summary statistics of selected characteristics by sex

	(1)	(2)
	Women	Men
dum_phstat1	0.28 (0.451)	0.28 (0.449)
dum_phstat2	0.32 (0.468)	0.32 (0.466)
dum_phstat3	0.27 (0.445)	0.27 (0.446)
dum_phstat4	0.09 (0.289)	0.10 (0.294)
dum_phstat5	0.03 (0.162)	0.03 (0.169)
dum_phstat6	0.00 (0.0352)	0.00 (0.0239)
dum_phstat7	0.00 (0.0218)	0.00 (0.0183)
dum_notcov1	0.14 (0.351)	0.15 (0.353)
dum_notcov2	0.85 (0.357)	0.85 (0.358)
dum_notcov3	0.01 (0.0745)	0.01 (0.0709)
N	20982	20982

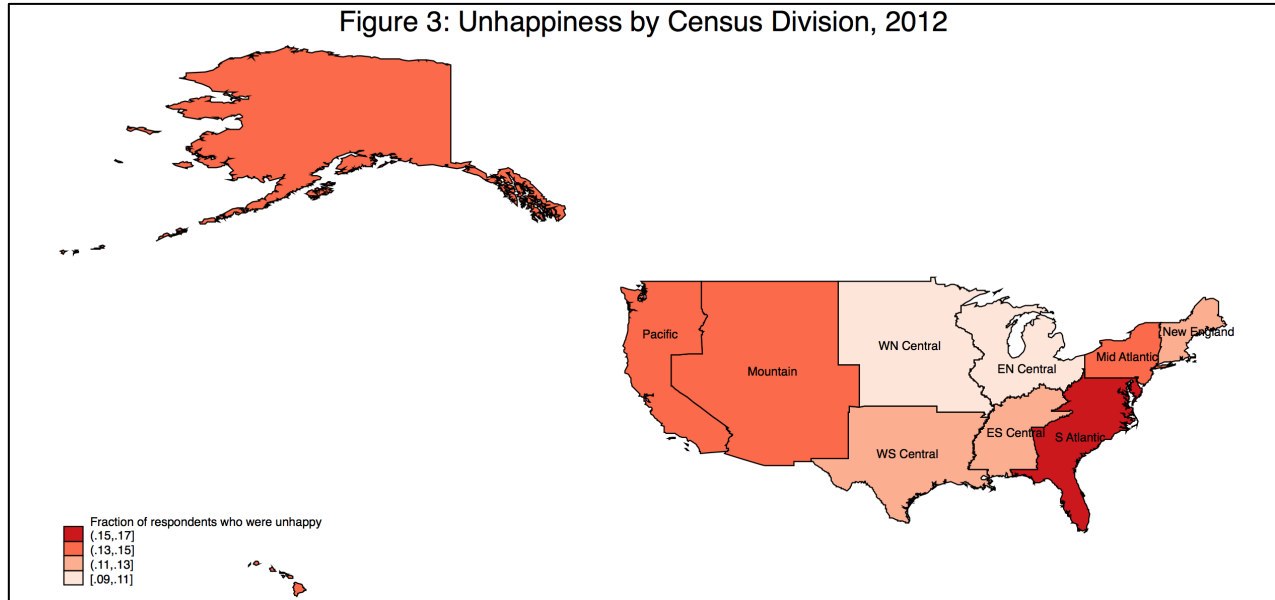
mean coefficients; sd in parentheses

Sample consists of matched husbands and wives from NHIS 2012.

The complete do-file for this exercise is here: http://www3.nd.edu/~jng2/workshop/_do-files/nhiscouples.do

8. EXERCISE 8

The goal of this exercise is to demonstrate Stata's mapping capabilities. For a serious GIS project, you should not be using Stata, but nevertheless Stata is capable of producing basic thematic maps. We will produce the map shown in **Figure 3**.



In general, to produce a thematic map, you need two ingredients:

- Data on the theme of interest. This dataset must contain a variable that identifies geographical location. For this tutorial, we need happiness data for 2012 at the Census Division-level. The dataset we use is **GSS2012_divisions.dta** http://www3.nd.edu/~jng2/workshop/_mapping/GSS2012_divisions.dta. You will note that the variable `subreg_id` identifies Census Divisions. The data are from the General Social Survey.
- A “shapefile” (a shapefile is actually a set of six files that collectively form a GIS map). This is basically the digital map on which you will overlay your theme of interest. We will use a shapefile provided by the U.S. Census Bureau.

STEP 1

The shapefile that you need is in **shapefile.zip** http://www3.nd.edu/~jng2/workshop/_mapping/shapefile.zip. Download it to your hard drive and unzip it to a folder of your choice, preferably the same one in which the necessary do-files and datasets (see the next steps) will reside.

STEP 2

You will need to convert the shapefile to

- a Stata-format dataset containing information from the original dBase file that is associated with the shapefile, and
- a Stata-format dataset containing geographical coordinates.

You do this using the add-on command `shp2dta`. Refer to `mapping1.do` for the complete command involved in this step: http://www3.nd.edu/~jng2/workshop/_do-files/makemap1.do

STEP 3

Merge **GSS2012_divisions.dta** and the dBase dataset, **usdata.dta** that was produced by shp2dta (see Step 2a).

Finally, the spmap command ties this data to the coordinates data produced by shp2dta (see Step 2b) and generates the map you see in Figure 3. The basic command is

```
spmap unhappy using uscoord.dta, id(subreg_id)
```

Refer to mapping2.do for the complete commands in this step:

http://www3.nd.edu/~jng2/workshop/_do-files/makemap2.do