3 Organizing and handling economic data

This chapter discusses four organizational schemes for economic data: the cross section, the time series, the pooled cross section/time series, and the panel dataset. Section 5 presents some tools for manipulating and summarizing panel data. Sections 6–8 present several Stata commands for combining and transforming datasets: `append`, `merge`, `joinby`, and `reshape`. The last section discusses using do-files to produce reproducible research and automatically validate data.

3.1 Cross-sectional data and identifier variables

A common type of data encountered in applied economics and finance is known as cross-sectional data, which contain measurements on distinct individuals at a given point in time. Those observations (rows in the Data Editor) vary over the units, such as individuals, households, firms, industries, cities, states, or countries. The variables (columns in the Data Editor) are generally measurements taken in a given period, such as household income for 2003, firms' reported profits for the first quarter of 2004, or cities' population in the 2000 Census. However, variables may contain measurements from other periods. For instance, a cross-sectional dataset of cities might contain variables named `pop1970`, `pop1980`, `pop1990`, and `pop2000` containing the cities' populations for those four decennial censuses. But unlike in time-series data, the observations in a cross-sectional dataset are indexed with an i subscript, without reference to t (the time).

In a cross-sectional dataset, the order of the observations in the dataset is arbitrary. We could `sort` the dataset on any of its variables to display or analyze extreme values of that variable without changing the results of statistical analyses, which implies that we can use Stata's `by` *varlist*: prefix. As discussed in section 2.1.8, using a `by` *varlist*: prefix requires that the data be sorted on the defined by-group, which you can do easily by using the `sort` option of the `by` *varlist*: prefix; that is, type `by` *varlist*`, sort:`. Time series, on the other hand, must follow a chronological order to be analyzed meaningfully.

Cross-sectional datasets usually have an identifier variable, such as a survey ID assigned to each individual or household, a firm-level identifier (e.g., a CUSIP code), industry-level identifier (e.g., a two-digit Standard Industrial Classification [SIC]) code, or a state or country identifier (e.g., MA, CT, US, UK, FRA, GER). Often there will be more than one identifier per observation. For instance, a survey might contain both a

household ID variable and a state identifier. Since Stata's variables may be declared as either numeric or string data types, practically any identifier available in an external data file may be used to define an identifier variable in Stata.

3.2 Time-series data

Cross-sectional datasets are found most often in applied microeconomic analysis. For example, a dataset might contain the share prices of the Standard and Poor's (S&P) 500 firms at the market close on a given day: a pure cross section. But we might also have a dataset that tracks a particular firm's share price, or that share price and the S&P index, daily for 2000–2003. The latter is a time-series dataset, and each observation would be subscripted by t rather than i. A time series is a sequence of observations on a given characteristic observed at a regular interval, such as x_t, x_{t+1}, $x_{t+\tau}$, with each period having the same length (though not necessarily an equal interval of clock time). The share price on the last trading day of each month may be between 26 and 31 days later than its predecessor (given holidays). For business-daily data, such as stock market prices, Friday is (usually) followed by Monday. But say that you had a list of dates and workers' wage rates, which records the successive jobs held and wages earned at the arbitrary intervals when workers received raises or took a new job. Those data could be placed on a time-series calendar, but they are not time-series data.

Periods in time-series data are identified by a Stata date variable, which can define annual, semiannual, quarterly, monthly, weekly, daily, or generic (undated) time intervals.[1] You can use `tsset` to indicate that this date variable defines the time-series calendar for the dataset. A nongeneric date variable should have one of the date formats (e.g., `%tq` for quarterly or `%td` for daily) so that dates will be reported as calendar units rather than as integers.

A few of Stata's time-series commands cannot handle *gaps*, or missing values, in the sequence of dates. Although an annual, quarterly, monthly, or weekly series might not contain gaps, daily and business-daily series often have gaps for weekends and holidays. You need to define such series in business days; for example, you could define a variable equal to the observation number (`_n`) as the date variable.

You must define a time-series calendar with `tsset` before you can use Stata's time-series commands (and some functions). But even if you do not need a time-series calendar, you should define such a calendar when you transform data so that you can refer to dates in doing statistical analysis: for instance, you may use `generate`, `summarize`, or `regress` with the qualifier `if tin(`*firstdate*`,`*lastdate*`)`. This function—which should be read as "t in"—lets you specify a range of dates, rather than observation numbers using a more cumbersome `in` *range*. The interval from the beginning to a specified date or from a date to the end of the sample may be given as `if tin(,`*lastdate*`)` or `if tin(`*firstdate*`,)`, respectively.

1. Defining the observation number (`_n`) as the date variable is the most often used generic time interval.

3.2.1 Time-series operators

Stata provides *time-series operators*—`L.`, `F.`, `D.`, `S.`—which let you specify *lags*, *leads* (forward values), *differences*, and *seasonal differences*, respectively. The time-series operators make it unnecessary to create a new variable to use a lag, difference, or lead. When combined with a *numlist*, they let you specify a set of these constructs in one expression. Consider the lag operator, `L.`, which when prepended to a variable name refers to the (first-) lagged value of that variable: `L.x`. A number may follow the operator so that `L4.x` would refer to the fourth lag of `x`—but more generally, a *numlist* may be used so that `L(1/4).x` refers to the first through fourth lags of `x` and `L(1/4).(x y z)` defines a list of the first through fourth lagged values of each of the variables `x`, `y`, and `z`. These expressions may be used anywhere that a *varlist* is required.

Like the lag operator, the lead operator `F.` lets you specify future values of one or more variables. The lead operator is unnecessary, since a lead is a negative lag, and an expression such as `L(-4/4).x` will work, labeling the negative lags as leads. The difference operator, `D.`, generates differences of any order. The first difference, `D.x`, is Δx or $x_t - x_{t-1}$. The second difference, `D2.x`, is not $x_t - x_{t-2}$, but rather $\Delta(\Delta x_t)$: that is, $\Delta(x_t - x_{t-1})$ or $x_t - 2x_{t-1} + x_{t-2}$. You can also combine the time-series operators so that `LD.x` is the lag of the first difference of `x` (that is, $x_{t-1} - x_{t-2}$) and refers to the same expression, as does `DL.x`. The seasonal difference `S.` computes the difference between the value in the current period and the period 1 year ago. For quarterly data, `S4.x` generates $x_t - x_{t-4}$, and `S8.x` generates $x_t - x_{t-8}$.

In addition to being easy to use, time-series operators will also never misclassify an observation. You could refer to a lagged value as `x[_n-1]` or a first difference as `x[_n] - x[_n-1]`, but that construction is not only cumbersome but also dangerous. Consider an annual time-series dataset in which the 1981 and 1982 data are followed by the data for 1984, 1985, ..., with the 1983 data not appearing in the dataset (i.e., not recorded as missing values, but physically absent). The observation-number constructs above will misinterpret the lagged value of 1984 to be 1982, and the first difference for 1984 will incorrectly span the 2-year gap. The time-series operators will not make this mistake. Since `tsset` has been used to define `year` as the time-series calendar variable, the lagged value or first difference for 1984 will be properly coded as missing, whether or not the 1983 data are stored as missing in the dataset.[2] Thus you should always use time-series operators when referring to past or future values or computing differences in a time-series dataset.

3.3 Pooled cross-sectional time-series data

Microeconomic data can also be organized into *pooled cross-section time series*, in which every observation has both an i and t subscript.[3] For example, we might have the responses from 3 weeks' presidential popularity polls in which each poll contains 400

2. The time-series operators also provide a similar benefit in panel data, as discussed below.
3. Econometricians often call data with this structure *pseudopanel data*; see Baltagi (2001).

randomly selected respondents. But the randomly sampled individuals who respond to the poll one week will probably not appear in the following poll or in any other poll drawn from a national sample before the election. These data are pooled cross sections over time such that observation j at time 1 has no relation to observation j at time 2 or time 3. We may use `collapse` to compute summary statistics for each cross section over time. For instance, if we have annual data for several random samples of U.S. cities for 1998–2004, we could use

```
. collapse (mean) income (median) medinc=income (sum) population, by(year)
```

which would create a new dataset with 1 observation per year, containing the year, average income, median income, and total population of cities sampled in that year.[4]

Although pooled cross-section/time-series data allow us to examine both the individual and time dimensions of economic behavior, they cannot be used to trace individuals over time. In that sense, they are much less useful than *panel* or *longitudinal* data, which I will now describe.

3.4 Panel data

A common form of data organization in microeconomics, macroeconomics, and finance is a type of pooled cross-sectional time-series data called *panel* or *longitudinal* data. *Panel data* contain measurements on the same individuals over several periods.[5] Perhaps the most celebrated longitudinal study of households is the University of Michigan's *Panel Study of Income Dynamics* (PSID), an annual survey of (originally) 5,000 households carried out since 1968. On the financial side, S&P COMPUSTAT databases of firm-level characteristics are one of the most important sources of panel data for financial research.

In this form of data organization, each individual's observations are identified, allowing you to generate microlevel measures not present in the original data. For example, if we have a pooled cross-sectional time-series dataset gathered from repeated annual surveys of randomly sampled individuals that measure their financial wealth along with demographics, we may calculate only an average net savings rate (the rate at which wealth is being accumulated or decumulated) or an average for subsamples of the data (such as the savings rate of African American respondents or of women under 35). We cannot monitor individual behavior, but if we have panel data on a group of individuals who have responded to annual surveys over the same time span, we can calculate individual savings rates and cohort measures for subsamples.

A panel dataset may be either *balanced* or *unbalanced*. In a balanced panel, each of the units, $i = 1, \ldots, G$, is observed in every period $t = 1, \ldots, T$, resulting in $G \times T$ observations in the dataset. Such a panel is easy to work with because the first T

4. The full syntax of `collapse` is described in section 3.4.

5. Panel data are so called because the first examples of these data were the time series of responses of a panel of experts, such as economic forecasters predicting next year's GDP growth or inflation rate. Panel data are also known as *longitudinal* data since they allow you to longitudinally analyze a cohort of households, workers, firms, stocks, or countries.

observations will correspond to unit 1, the second T to unit 2, and so on. However, economic and financial data are often not available in balanced form because some individuals drop out of a multiyear survey.

Furthermore, if we constrain analysis to a balanced panel, we create *survivorship bias*. For example, the S&P COMPUSTAT database of U.S. firms contains 20 years of annual financial statement data—but only for those firms that have existed for the entire period. The set of firms is thus unrepresentative in omitting startups (even those of age 19) and firms that were taken over during that time. Although the algebra of panel-data transformations and estimators is simplified with a balanced panel, I often prefer to work with an unbalanced panel to avoid such biases and mitigate the loss of sample size that may result from insisting on balanced-panel data.

Stata's tools for panel data make it easy to work with any set of observations, balanced or unbalanced, that can be uniquely identified by i and t. Unlike Stata, many statistical packages and matrix languages require a balanced structure with T observations on each of G units, even if some of them are wholly missing.[6] You can use `tsset` to indicate that the data are panel data. The same command that defines a time-series calendar for a time series may specify the panel variable as well:

`tsset` *panelvar timevar*

The *timevar* must be a date variable, whereas *panelvar* may be any integer variable that uniquely identifies the observations belonging to a given unit. The integer values need not be sequential: that is, we could use three-digit SIC codes `321`, `326`, `331`, and `342` as the *panelvar*. But if the units of the data are identified by a string variable, such as a two-letter state abbreviation, we must `encode` that variable to create a *panelvar* identifier. `tsset` will report the ranges of *panelvar* and *timevar*.

3.4.1 Operating on panel data

Stata contains a thorough set of tools for transforming panel data and estimating the parameters of econometric models that take account of the nature of the data. Any `generate` or `egen` functions that support a `by` *varlist*: may be applied to panel data by using the *panelvar* as the *by-variable*. Descriptive data analysis on panel data often involves generating summary statistics that remove one of the dimensions of the data. You may want to compute average tax rates across states for each year or average tax rates over years for each state. You can compute these sets of summary statistics by using the `collapse` command, which produces a dataset of summary statistics over the elements of its `by(`*varlist*`)` option. The command syntax is

6. You can use Stata's `tsfill` command to generate such a structure from an unbalanced panel.

`collapse` *clist* [*if*] [*in*] [*weight*] [, *options*]

where the *clist* is a list of [(*stat*)] *varlist* pairs or a list of [(*stat*)] *target_var*=*varname* pairs. In the first format, the *stat* may be any of the descriptive statistics available with `summarize` (see [R] **summarize**), with some additions: e.g., all 100 percentiles may be computed. If not specified, the default *stat* is `mean`. To compute more than one summary statistic for a given variable, use the second form of the command, where the *target_var* names the new variable to be created. The `by(`*varlist*`)` option specifies that `collapse` generate one result observation for each unique value of the `by(`*varlist*`)`. For more information on the `collapse` syntax, see [D] **collapse**.

The `grunfeld` dataset contains annual firm-level data for 10 U.S. firms over 20 years, as the output from `tsset` shows. We first `summarize` three variables over the entire panel:

```
. use http://www.stata-press.com/data/imeus/grunfeld, clear
. tsset
        panel variable:  company, 1 to 10
         time variable:  year, 1935 to 1954
. summarize mvalue invest kstock
    Variable |       Obs        Mean    Std. Dev.       Min        Max
-------------+--------------------------------------------------------
      mvalue |       200    1081.681     1314.47     58.12     6241.7
      invest |       200    145.9583    216.8753       .93     1486.7
      kstock |       200    276.0172    301.1039        .8     2226.3
```

After using `preserve` to retain the original data, we use `collapse` by `year` to generate the mean market value, sum of investment expenditures, and mean of firms' capital stock for each year.

```
. preserve
. collapse (mean) mvalue (sum) totinvYr=invest (mean) kstock, by(year)
. graph twoway tsline mvalue totinvYr kstock
```

I plot these time-series data in figure 3.1 to illustrate that the cross-sectional summary statistics trend upward over these two decades.[7]

7. I present Stata graphics in this text without explaining the syntax of Stata's graphics language. For an introduction to Stata graphics, see `help graph intro` and [G] **graph intro**. For an in-depth presentation of Stata's graphics capabilities, see *A Visual Guide to Stata Graphics* (Mitchell 2004).

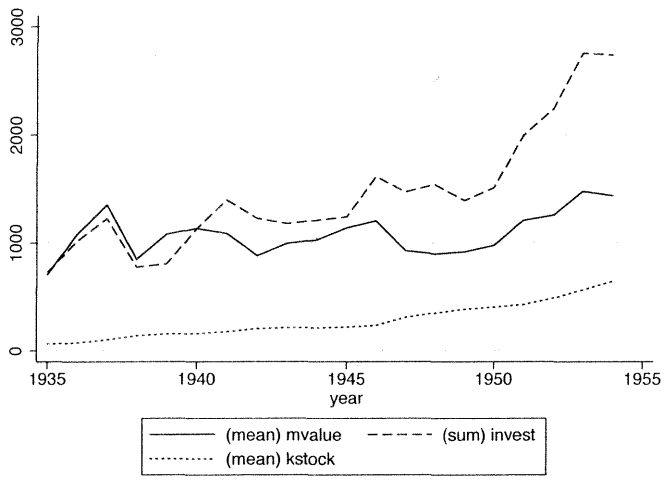

Figure 3.1: Graph of panel data collapsed to time series

When performing data transformations on panel data, you should take advantage of the time-series operators' housekeeping facilities. Consider the dataset above. If the lagged value of `mvalue` is generated with `mvalue[_n-1]`, you must explicitly exclude the first observation of each firm from the computation. Otherwise its lagged value would refer to the last observation of the prior firm for firms 2, ..., 10. In contrast, you can use

```
. generate lagmvalue = L.mvalue
```

without considering the panel nature of the data. Each firm's first observation of `lagmvalue` will be defined as missing.

Stata's commands for panel data are described in [XT] **xt** and [XT] **intro**. Each command's name begins with `xt`. Chapter 9 introduces some estimation techniques for panel data in economic analysis.

3.5 Tools for manipulating panel data

Section 3.4 introduces balanced and unbalanced panel (longitudinal) data and shows how to use the `collapse` command to create a pure time series or a pure cross section from panel data. As described in section 3.4, you should always use `tsset` to set up panel data so that you can use Stata's time-series operators and `xt` commands. Sometimes translating an external date format into Stata's date variables is cumbersome. Say that you import a time series, perhaps with a spreadsheet-formatted date variable, and want to establish a time-series calendar for these data. You must work with the existing date to make it a Stata date variable by using the `date()` or `mdy()` functions, assign a format

to the variable (e.g., %ty for annual, %tm for monthly), and then use tsset to define the time-series calendar with that formatted date variable. You can use the tsmktim utility to do all these steps together (Baum and Wiggins 2000). You need only specify the name of a new time-series calendar variable and its start date:

```
. tsmktim datevar, start(1970)
```

A new version of the routine (available from ssc) allows you to generate a date variable for each unit within a panel:

```
. tsmktim datevar, start(1970q3) i(company)
```

Each unit's series must start (but need not end) in the same period.

3.5.1 Unbalanced panels and data screening

Researchers organizing panel data often apply particular conditions to screen data. For instance, unbalanced firm-level panel data may have only one or two annual observations available for particular firms rather than the 20 years available for more-established firms in the sample. Here I discuss several commands that you can use before estimation to manipulate panel data organized in the *long* format (see section 3.6).

You can use the xtdes command to describe the pattern of panel data, particularly to determine whether the panel is balanced or unbalanced.[8] This command leaves scalars in r(): r(N) gives the number of panel units, with r(min) and r(max) giving the minimum and maximum number of observations available per unit. If those two items are equal, the panel is balanced. As discussed earlier, you could use tsfill to "rectangularize" an unbalanced panel by filling in missing observations with missing values. But what if you wanted to remove any panel units with fewer than the maximum number of observations? You could create a new variable counting observations within each unit as a by-group and use that variable to flag units with missing observations:

```
. xtdes
. local maxobs = r(max)
. by company: generate obs = _N
. drop if obs < `maxobs'
```

Often an unbalanced panel is preferable, but we may want to screen out units that have fewer than m observations; if $m = 10$, for instance, drop if obs < 10 would remove units failing to pass that screen.

Using this logic will ensure that each firm has the minimum number of observations, but it cannot guarantee that they are without gaps. Some of Stata's time-series estimation and testing routines do not allow gaps within time series, although the tsset command saves macros r(tmins) and r(tmaxs) to signal the first and last periods of any unit of the panel.

8. It is tempting to imagine that this command is named xtdesc, but it is not.

Nicholas Cox's routine `tsspell` (available from `ssc`) identifies complete runs of data, or "spells", within a time series or within a panel of time series. A gap terminates the spell and starts a new spell following the gap. Thus `obs == 'maxobs'` would correspond to a unit with a spell of `'maxobs'`. The routine is general and may be used to identify spells on the basis of a logical condition (for instance, the sign of a variable such as GDP growth changing from positive to negative or a variable identifying the party in power changing). We will use a simpler aspect of the routine to identify gaps in the time series as shown by the calendar variable.

Consider the missing data in a modified version of the *Stata Longitudinal/Panel Data Reference Manual* `grunfeld` dataset. The original dataset contains a balanced panel of 20 years of annual data on 10 firms. In the modified version, five of those firms lack one or more observations: one firm "starts late", one firm "ends early", and three firms' series have embedded gaps:

```
. use http://www.stata-press.com/data/imeus/grunfeldGaps, clear
. xtdes
 company:  1, 2, ..., 10                                    n =          10
    year:  1935, 1936, ..., 1954                            T =          20
           Delta(year) = 1; (1954-1935)+1 = 20
           (company*year uniquely identifies each observation)
Distribution of T_i:   min      5%     25%       50%       75%     95%     max
                        17      17      18        20        20      20      20
     Freq.  Percent    Cum. | Pattern
------------------------------+---------------------
         5     50.00   50.00 | 11111111111111111111
         1     10.00   60.00 | ..1111111111111111111
         1     10.00   70.00 | 111111111.1111.11111
         1     10.00   80.00 | 111111111111...11111
         1     10.00   90.00 | 1111111111111.111111
         1     10.00  100.00 | 11111111111111111..
------------------------------+---------------------
        10    100.00         | XXXXXXXXXXXXXXXXXXXX
```

We identify these conditions by using `tsspell` with the condition `D.year == 1`.[9] For series with gaps, that condition will fail. The `tsspell` routine creates three new variables, `_spell`, `_seq`, and `_end`, and we are concerned with `_spell`, which numbers the spells in each firm's time series. A firm with one unbroken spell (regardless of starting and ending dates) will have `_spell = 1`. A firm with one gap will have later observations identified by `_spell = 2`, and so on. To remove all firms with embedded gaps, we may apply similar logic to that above to drop firms with more than one reported spell:

```
. tsspell year, cond(D.year == 1)
. egen nspell = max(_spell), by(company)
. drop if nspell > 1
(54 observations deleted)
```

9. This condition would work for any other Stata data frequency, since half-years, quarters, months, weeks, and days are also stored as successive integers.

```
. xtdes
 company:  2, 3, ..., 10                                    n =          7
    year:  1935, 1936, ..., 1954                            T =         20
           Delta(year) = 1; (1954-1935)+1 = 20
           (company*year uniquely identifies each observation)
Distribution of T_i:   min      5%     25%       50%      75%      95%     max
                        18      18      18        20       20       20      20
     Freq.  Percent    Cum. |  Pattern
----------------------------+----------------------
         5    71.43   71.43 |  11111111111111111111
         1    14.29   85.71 |  ..11111111111111111111
         1    14.29  100.00 |  11111111111111111...
----------------------------+----------------------
         7   100.00         |  XXXXXXXXXXXXXXXXXXXX
```

Or if we were willing to retain firms with gaps but did not want to keep any spell shorter than a certain length (say, 5 years) we could use the `_seq` variable, which counts the length of each spell, and include `_spell` in the `egen` command that computes the maximum over each firm and spell within the firm's observations:

```
. use http://www.stata-press.com/data/imeus/grunfeldGaps, clear
. tsspell year, cond(D.year == 1)
. replace _spell = F._spell if _spell == 0
(14 real changes made)
. egen maxspell = max(_seq+1), by(company _spell)
. drop if maxspell < 5
(4 observations deleted)
. xtdes
 company:  1, 2, ..., 10                                    n =         10
    year:  1935, 1936, ..., 1954                            T =         20
           Delta(year) = 1; (1954-1935)+1 = 20
           (company*year uniquely identifies each observation)
Distribution of T_i:   min      5%     25%       50%      75%      95%     max
                        14      14      18        20       20       20      20
     Freq.  Percent    Cum. |  Pattern
----------------------------+----------------------
         5    50.00   50.00 |  11111111111111111111
         1    10.00   60.00 |  ..11111111111111111111
         1    10.00   70.00 |  111111111......11111
         1    10.00   80.00 |  111111111111...11111
         1    10.00   90.00 |  1111111111111.111111
         1    10.00  100.00 |  11111111111111111...
----------------------------+----------------------
        10   100.00         |  XXXXXXXXXXXXXXXXXXXX
```

The resulting dataset includes firms' spells of 5 years or more. The `_spell` variable is recoded from 0 to its following value; by default, `tsspell` places a zero in the first observation of each spell, but we want that value to be the current spell number. Likewise, because the `_seq` variable starts counting from zero, we consider the maximum value of (`_seq + 1`) to evaluate the spell length. Experimenting with `tsspell` will reveal its usefulness in enforcing this type of constraint on the data.

3.5.2 Other transforms of panel data

Some analyses require smoothing the data in each panel; `tssmooth` (see [TS] **tssmooth**) provides the most widely used smoothers, all of which can be applied to the data in each panel. For example, we might want a weighted moving average of four prior values, with arithmetic weights `0.4(0.1)0.1`. That construct can be viewed as a filter applied to a series in the time domain and computed with `tssmooth ma`:

```
. tssmooth ma wtavg = invest, weights(0.1(0.1)0.4 <0>)
```

The weights are applied to the fourth, third, second, and first lags of `invest`, respectively, to generate the variable `wtavg`. The `<0>` is a placeholder to instruct Stata that the zero-lag term should be given a weight of zero. This command can also be used to impose a two-sided filter with varying weights:

```
. tssmooth ma wtavg  = invest, weights(1 4 <6> 4 1)
```

This command specifies that a two-sided centered moving average be computed, with weights 1/16, 4/16, 6/16, 4/16, and 1/16. You can apply the `tssmooth ma` command to panel data because the filter is automatically applied separately to each time series within the panel.

Other analyses use functions of the extreme values in each series. For example, the `record()` `egen` part of Nicholas Cox's `egenmore` package (available from `ssc`) provides one solution. For example,

```
. egen maxtodate = record(wage), by(id) order(year)
. egen hiprice = record(share_price), by(firm) order(quote_date)
```

The first example identifies the highest wage to date in a worker's career, whereas the second identifies the highest price received to date for each firm's shares.

3.5.3 Moving-window summary statistics and correlations

When working with panel data, you often want to calculate summary statistics for subperiods of the time span defined by the panel calendar. For instance, if you have 20 years' data on each of 100 firms, you may want to calculate 5-year averages of their financial ratios. You can calculate these averages with the `tabstat` (see [R] **tabstat**) command. You need only define the 5-year periods as the elements of a by-group and specify that selector variable as the arguments to `tabstat`'s `by()` option, while prefixing the command with `by firmid:`. However, to retrieve the computed statistics, you will need to use the `save` option, which stores them in several matrices. Or you could use several `egen` statements to generate these statistics as new variables, using the same by-group strategy.

To compute summary statistics from *overlapping* subsamples, we could define a by-group, but here Stata's `by` capabilities cannot compute statistics from a sequence of by-groups that are formed by a "moving window" with, for example, 11 months'

overlap. The mvsumm routine of Baum and Cox (available from [R] **ssc**) computes any of the univariate statistics available from summarize, detail and generates a time series containing that statistic over the defined time-series sample. You can specify the window width (the number of periods included in the statistic's computation) as an option, as well as the alignment of the resulting statistic with the original series. This routine is especially handy for financial research, in which some measure of recent performance—the average share price over the last 12 months or the standard deviation (volatility) of the share price over that interval—is needed as a regressor. The mvsumm routine will operate separately on each time series of a panel, as long as a panel calendar has been defined with tsset.

Another way to generate moving-window results is to use Stata's rolling prefix, which can execute any statistical command over moving windows of any design. However, rolling is more cumbersome than mvsumm, since it creates a new dataset containing the results, which then must be merged with the original dataset.

To calculate a moving correlation between two time series for each unit of a panel, you can use Baum and Cox's mvcorr routine (available from ssc). This computation is useful in finance, where computing an optimal hedge ratio involves computing just such a correlation, for instance, between spot and futures prices of a particular commodity. Since the mvcorr routine supports time-series operators, it allows you to compute moving autocorrelations. For example,

```
. mvcorr invest L.invest, win(5) gen(acf) end
```

specifies that the first sample autocorrelation of an investment series be computed from a five-period window, aligned with the last period of the window (via option end), and placed in the new variable acf. Like mvsumm, the mvcorr command operates automatically on each time series of a panel:[10]

```
. use http://www.stata-press.com/data/imeus/grunfeld, clear
. drop if company>4
(120 observations deleted)
. mvcorr invest mvalue, window(5) generate(rho)
. xtline rho, yline(0) yscale(range(-1 1))
> byopts(title(Investment vs. Market Value: Moving Correlations by Firm))
```

Figure 3.2 shows the resulting graph of four firms' investment-market value correlations.

10. For a thorough presentation of the many styles of graphs supported by Stata, see Mitchell (2004).

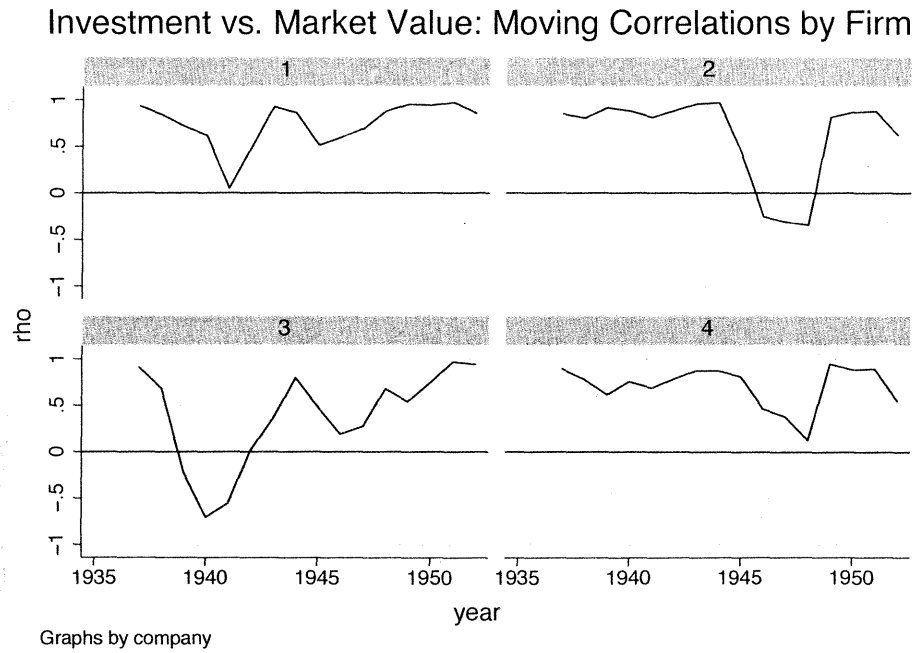

Figure 3.2: Moving-window correlations

3.6 Combining cross-sectional and time-series datasets

Applied economic analysis often involves combining datasets. You may want to pool the data over different cross-sectional units or build a dataset with both cross-sectional and time-series characteristics. In the former case, you may have 200 observations that reflect probable voters' responses to a telephone survey carried out in Philadelphia, 300 observations from that same survey administered in Chicago, and 250 observations from voters in Kansas City. In the latter case, you may have a dataset for each of the six New England states containing annual state disposable personal income and population for 1981–2000. You may want to combine those six datasets into one dataset. How you combine them—over cross sections or over the cross-section and time-series dimensions—depends on the type of analysis you want to do. We may want to work with data in what Stata calls *wide format*, in which measurements on the same variable at different points in time appear as separate variables. For instance, we might have time series of population for each of the New England states, with variables named `CTpop`, `MApop`, `MEpop`, ... In contrast, you may find it easier to work with *long-format* data, in which those same data are *stacked*, so that you have one variable, `pop`, with sets of observations associated with each state. You then must define another variable that identifies the

unit (here the state). Stata has commands for each type of combination, as well as the `reshape` command for transforming data between the wide and long formats. We first discuss combining cross-sectional datasets to produce a pooled dataset in the long format.

3.7 Creating long-format datasets with append

If we have the three voter survey datasets for Philadelphia, Chicago, and Kansas City mentioned above and want to combine them into one pooled dataset, we must first ensure that each dataset's variable names are identical. We will use the `append` command, and since `prefBush` and `prefBUSH` are different variables to Stata, we cannot combine files containing those variables and expect them to properly align. We can use `rename` to ensure that variable names match. We will also want to be able to recover the city identifier from the combined dataset, even if it is not present in the individual datasets. We could remember that the first 200 observations come from Philadelphia, the next 300 from Chicago, and so on, but if the dataset is ever sorted into a different order, we will lose this identifier. Thus we should insert a new variable, `city`, into each dataset, which could be either a numeric variable with a value label of the city's name or a string variable that can be `encoded` into numeric form for use in a `by` *varlist*:. We can then use `append` to combine them:

```
. use philadelphia, clear
. append using chicago
. append using kcity
. save vote3cities, replace
```

If we have a string variable, `city`, containing the city name, we can use

```
. encode city, gen(citycode)
```

to create a numeric city identifier. We could then use the `citycode` variable in a statistical command or to create a set of city-specific dummy variables with the `tabulate` command's `generate()` option; see [R] **tabulate oneway**. Although our example illustrates how three datasets can be combined, any number of datasets could be combined in this manner.

This dataset is in Stata's long format; for each variable, the measurements for each unit within the panel are stored in separate observations. Since we have combined three pure cross sections, each of arbitrary order, there is no relationship among respondent #1 in Philadelphia, respondent #1 in Chicago, and respondent #1 in Kansas City. But this long format makes it easy to do many computations that would be cumbersome if the data were combined any other way. For instance, you can easily compute whether Philadelphia's mean of `prefBush` is equal to Chicago's or whether all three means of `prefBush` are statistically distinguishable. That would not be the case if the three datasets' variables were recombined horizontally rather than vertically.

3.7.1 Using merge to add aggregate characteristics

The long-format dataset we constructed above is useful if we want to add aggregated information to individual records. For instance, imagine that the voter survey data contain each individual's five-digit ZIP code (`zipcode`) along with his or her preferences for the presidential candidates, and we want to evaluate whether the voter's income level as proxied by the average income in her ZIP code affects her voting preferences. How may we append the income information to each record? We could use a sequence of `replace` statements or a complicated nested `cond()` function. But you can easily create a new dataset containing ZIP codes (in the same five-digit integer form) and average income levels. If these data are acquired from the U.S. Census, you should have records for every Philadelphia (or Chicago, or Kansas City) ZIP code or for the entire state in which each city is located. But you will be using this file merely as a lookup table. We can then sort the dataset by ZIP code and save it as `incbyzip.dta`. We can combine this information with the original file by using the following commands:

```
. use vote3cities, clear
. sort zipcode
. merge zipcode using incbyzip, nokeep
```

Using `merge` in this way is known as a *one-to-many* match-merge where the income for each ZIP code is added to each voter record in that ZIP code. The `zipcode` variable is the *merge key*. Both the *master file* (`vote3cities.dta`) and the *using file* (`incbyzip.dta`) must be sorted by the merge key. By default, `merge` creates a new variable, `_merge`, which takes on an integer value of 1 if that observation was found only in the master dataset, 2 if it was found only in the using dataset, or 3 if it was found in both datasets. Here we expect `tab _merge` to reveal that all values equal 3. Each voter's ZIP code should be mapped to a known value in the using file. Although many ZIP codes in the using file may not be associated with any voter in the sample, which would yield a `_merge` of 2, we specified the `nokeep` option to drop the unneeded entries in the using file from the merged file. We could then use

```
. assert _merge == 3
. drop _merge
```

to verify that the match was successful. Using `merge` is much easier than using a long and complicated do-file that uses `replace`. By merely modifying the `using` file, we can correct any problems in the one-to-many merge. If we had several ZIP code–specific variables to add to the record, such as average family size, average proportion minority, and average proportion of home ownership, we could handle these variables with one `merge` command. This technique is useful for working with individual data for persons, households, plants, firms, states, provinces, or countries when we want to combine the microdata with aggregates for ZIP codes, cities, or states at the individual level; industries or sectors at the plant or firm level; regions or the macroeconomy at the state or province level; and global regions or world averages at the country level.

3.7.2 The dangers of many-to-many merges

You can also use the `merge` command to combine datasets by using a *one-to-one* match-merge. For example, we might have two or more datasets whose observations pertained to the same units: e.g., U.S. state population figures from the 1990 and 2000 Censuses.

The many-to-many merge is a potential problem that arises when there are multiple observations in both datasets for some values of the merge key variable(s). Match-merging two datasets that both have more than one value of the merge key variable(s) can cause repeated execution of the same do-file to have a different number of cases in the result dataset without indicating an error. A coding error in one of the files usually causes such a problem. You can use the `duplicates` command to track down such errors. To prevent such problems, specify either the `uniqmaster` or `uniqusing` option in a match-merge. For instance, the ZIP code data should satisfy `uniqusing` in that each ZIP code should be represented only once in the file. In a one-to-one match-merge, such as the state income and population data, you could use the `unique` option, as it implies both `uniqmaster` and `uniqusing` and asserts that the merge key be unique in both datasets.[11]

Beyond `append` and `merge`, Stata has one more command that combines datasets: `joinby` is used less often since its task is more specialized. The command creates a new dataset by forming all possible pairwise combinations of the two datasets, given the merge key. Usually, you will want to use `merge` instead of `joinby`.[12]

3.8 The reshape command

If your dataset is organized in long or wide format, you may need to reorganize it to more easily obtain data transformations, statistics, or graphics. To solve this problem, you can use the `reshape` (see [D] **reshape**) command, which reorganizes a dataset in memory without modifying data files. Some statistical packages do not have a reshape feature, so you would need to write the data to one or more external text files and read it back in. With `reshape`, this extra step is not necessary in Stata, but you must label the data appropriately. You may need to do some experimentation to construct the appropriate command syntax, which is all the more reason for using a do-file, since someday you are likely to come upon a similar application for `reshape`.

Consider a wide-format dataset with variables labeled `pop1970`, `pop1980`, `pop1990`, `pop2000`, and `area`, and with observations identified by each of the six New England state codes.

11. Those familiar with relational database-management systems, such as SQL, will recognize that *uniqueness* means that the merge key is a valid and unique *primary key* for the dataset.
12. Those familiar with relational database-management systems will recognize `joinby` as the SQL *outer join*: a technique to be avoided in most database tasks.

```
. use http://www.stata-press.com/data/imeus/reshapeState, clear
. list
```

	state	pop1970	pop1980	pop1990	pop2000	area
1.	CT	.1369841	.6184582	.4241557	.2648021	.871691
2.	MA	.6432207	.0610638	.8983462	.9477426	.4611429
3.	ME	.5578017	.5552388	.5219247	.2769154	.4216726
4.	NH	.6047949	.8714491	.8414094	.1180158	.8944746
5.	RI	.684176	.2551499	.2110077	.4079702	.0580662
6.	VT	.1086679	.0445188	.5644092	.7219492	.6759487

We want to reshape the dataset into long format so that each state's and year's population value will be recorded in one variable. We specify `reshape long pop` so that the variable to be placed in the long format will be derived from all variables in the dataset whose names start with `pop`. The command works with $x_{i,j}$ data; in Stata, i defines a panel, and j defines an identifier that varies within each panel. Here the state defines the panel, so we specify that `state` is the `i()` variable; here the `j()` option specifies that the suffixes of the `pop` variable be retained as a new variable, `year`:

```
. reshape long pop, i(state) j(year)
(note: j = 1970 1980 1990 2000)
Data                               wide   ->   long
-----------------------------------------------------------------------------
Number of obs.                        6   ->     24
Number of variables                   6   ->      4
j variable (4 values)                     ->   year
xij variables:
           pop1970 pop1980 ... pop2000    ->   pop
-----------------------------------------------------------------------------
```

The 6 observations of the original wide dataset have been expanded to 24, since each state had four population figures in the original form:

(*Continued on next page*)

```
. list
```

	state	year	pop	area
1.	CT	1970	.1369841	.871691
2.	CT	1980	.6184582	.871691
3.	CT	1990	.4241557	.871691
4.	CT	2000	.2648021	.871691
5.	MA	1970	.6432207	.4611429
6.	MA	1980	.0610638	.4611429
7.	MA	1990	.8983462	.4611429
8.	MA	2000	.9477426	.4611429
9.	ME	1970	.5578017	.4216726
10.	ME	1980	.5552388	.4216726
11.	ME	1990	.5219247	.4216726
12.	ME	2000	.2769154	.4216726
13.	NH	1970	.6047949	.8944746
14.	NH	1980	.8714491	.8944746
15.	NH	1990	.8414094	.8944746
16.	NH	2000	.1180158	.8944746
17.	RI	1970	.684176	.0580662
18.	RI	1980	.2551499	.0580662
19.	RI	1990	.2110077	.0580662
20.	RI	2000	.4079702	.0580662
21.	VT	1970	.1086679	.6759487
22.	VT	1980	.0445188	.6759487
23.	VT	1990	.5644092	.6759487
24.	VT	2000	.7219492	.6759487

In the wide format, the observations are labeled $i = 1, \ldots,$ `_N`, and each measurement to be transformed to the long form consists of variables indexed by $j = 1, \ldots, J$. The *varlist* for `reshape long` lists the *base names* or *stubs* of all variables that are in the $x_{i,j}$ form and should be `reshape`d to the long form. Here we have only `pop` with $J = 4$ because there are four decennial census years of population data; this same $x_{i,j}$ format may include several variables. For instance, our dataset might contain additional variables `popM1970`, `popF1970`, ..., `popM2000`, `popF2000` with gender breakdowns of state population. The `reshape long` statement's *varlist* would then read `pop popM popF` because these variables are to be treated analogously.

You must specify `j()`, as the long format requires the j identifier. Here the j dimension is the year, but it could be any characteristic. Instead of state population measures for different years, we could have `popWhite`, `popBlack`, `popHispanic`, and `popAsian`. Then we would use the options `j(race) string`, to specify that the j identifier is the `string` variable `race` (which you could then `encode`). Variables that do not vary over j (`year` or `race`) are not specified in the `reshape` statement. In the example above, the state's `area` is time invariant, so it is automatically replicated for each `year`.

We continue with the long-format dataset that results from the example above but now want the data in wide format. We then use `reshape wide` to specify that the `pop` variable be spread over the values of `j(year)`. The rows of the resulting wide-format dataset are defined by the `i(state)` option:

```
. reshape wide pop, i(state) j(year)
(note: j = 1970 1980 1990 2000)
Data                                long   ->   wide
-----------------------------------------------------------------------------
Number of obs.                        24   ->      6
Number of variables                    4   ->      6
j variable (4 values)               year   ->   (dropped)
xij variables:
                                     pop   ->   pop1970 pop1980 ... pop2000
-----------------------------------------------------------------------------
```

This command is the same as the `reshape long` in the prior example, with `long` replaced by `wide`. The same information is required: you must specify the variables to be widened (here named explicitly, not by stubs), the panel's i variable, and the within-panel identifier (j variable). In creating the wide-format data, the j variable is dropped because its values are now spread over the columns `pop1970`, `pop1980`, `pop1990`, and `pop2000`. To illustrate,

```
. list
```

	state	pop1970	pop1980	pop1990	pop2000	area
1.	CT	.1369841	.6184582	.4241557	.2648021	.871691
2.	MA	.6432207	.0610638	.8983462	.9477426	.4611429
3.	ME	.5578017	.5552388	.5219247	.2769154	.4216726
4.	NH	.6047949	.8714491	.8414094	.1180158	.8944746
5.	RI	.684176	.2551499	.2110077	.4079702	.0580662
6.	VT	.1086679	.0445188	.5644092	.7219492	.6759487

You need to choose appropriate variable names for `reshape`. If our wide dataset contained `pop1970`, `Pop1980`, `popul1990`, and `pop2000census`, you would not be able to specify the common stub labeling the choices. However, say that we have for each state the measures `pop1970`, `pop1970M`, and `pop1970F`. The command

```
. reshape long pop pop@M pop@F, i(state) j(year)
```

uses `@` as a placeholder for the location of the j component of the variable name. Similarly, in our `race` example, if the variables were named `Whitepop`, `Blackpop`, `Hispanicpop`, and `Asianpop`, the command

```
. reshape long @pop, i(state) j(race) string
```

would handle those names. In more difficult cases, where repeatedly using `rename` may be tedious, `renvars` (Cox and Weesie 2001) may be useful.

This discussion has only scratched the surface of `reshape`. See [D] **reshape** for more information, and experiment with the command.

3.8.1 The xpose command

You can use the `xpose` command to make radical changes to the organization of your data. This command turns observations into variables and vice versa. This functionality is common in spreadsheets and matrix languages, but it is rarely useful in Stata because applying `xpose` will usually destroy the contents of string variables. If all variables in the dataset are numeric, this command may be useful. Rather than using `xpose`, consider reading in the raw data with the `byvariable()` option of `infile` (see [D] **infile (free format)**). If you need to transpose the data, they were probably not created sensibly in the first place.

3.9 Using Stata for reproducible research

3.9.1 Using do-files

Stata's command-line syntax makes it easy to document your data transformations, statistical analyses, and graphics creation. For some users, the command line is a nuisance, so they applauded Stata's dialogs when they appeared in version 8. But even Stata's dialogs produce complete commands in the Review window.

Stata does not require you to keep a record of the commands you issued, but most research requires that you be able to reproduce findings. Unless you carefully document every step of the process, you will not be able to reproduce your findings later, which can be disastrous. With Stata, you can document your research by using a do-file.

A do-file contains a sequence of Stata commands and can be invoked by selecting File ▷ Do... from the menu, by double-clicking on its icon, or by issuing the `do` command at the command line. Normally, a do-file will stop if it encounters an error. You can construct a do-file by using the Do-file Editor in Stata or any text editor. A do-file is merely a text file. You can include comments in a do-file, as long as they follow Stata conventions. Including such comments can help you remember what you did in your research. Placing a creation or revision date in a do-file is good practice.

An entire research project can involve hundreds or thousands of Stata commands from start to finish, so including all of them in a massive do-file would be cumbersome. Instead, consider writing a *master* do-file that calls a sequence of do-files to perform each step of the process: data input, data validation and cleaning, data transformation, statistical analyses, and production of graphical and tabular output. Each step might in turn be carried out by several do-files. If you follow this strategy, then it becomes straightforward to redo some step of the analysis in case of error or to rebuild a `.dta` file if it is inadvertently modified.

This strategy of using modular do-files for each step of the research project works well when you need to conduct a parallel analysis on another dataset. Many survey datasets have annual waves. Your processing of the latest wave of the survey will probably follow many of the same steps as last year's. With a well-organized and well-documented set of do-files, you need only copy those files and apply them to the new set of data.

No software package will force you to be a responsible researcher, but Stata makes following good research habits easy so that you can return to a research project and see exactly what you did to generate the final results.

3.9.2 Data validation: assert and duplicates

Before you can effectively manage data, you need to make sure that all the values of the raw data make sense. Are there any apparent coding errors in the data values? Should any values of numeric variables properly be coded as some sort of missing data, as discussed above? As mentioned, to construct an audit trail for your data management, you can create a do-file that reads the raw data, applies several checks to ensure that data values are appropriate, and writes the initial Stata binary data file. This data file should not be modified in later programs or interactive analysis. Each program that uses the file and creates additional variables, subsets, or merges of the data should save the resulting modified file under a new name. Each step in the data validation and transformation process can then be documented and reexecuted as needed. Even if the raw data are provided in Stata binary format from an official source, you should assume that there are coding errors.

You should follow this method from the beginning of the data-management process. On Statalist, users often say such things as, "I did the original data transformations (or merges) in Excel, and now I need to ...." Even if you are more familiar with a spreadsheet syntax than with the Stata commands needed to replicate that syntax, you should use Stata so that you can document and reproduce its operations on the data. Consider two research assistants starting with the same set of 12 spreadsheets. They are instructed to construct one spreadsheet performing some complicated append or merge processes by using copy and paste. What is the probability that the two assistants will produce identical results? Probably less than one.

The proposed solution: export the 12 spreadsheets to text format, and read them into Stata by using a do-file that loops over the `.txt` or `.csv` files and applies the same transformations to each one, performing the appropriate `append` or `merge` operations. That do-file, once properly constructed, will produce a reproducible result. You can easily modify the do-file to perform a similar task, such as handling 12 spreadsheets containing cost elements rather than revenues. You can (and should) add comments to the do-file documenting its purpose, dates of creation/modification, and creator/modifier. You may either place an asterisk (`*`) at the beginning of each comment line, use the block comment syntax (`/*` to begin a comment, `*/` to end it) to add several lines of comments to a do-file, or use two forward slashes (`//`) to add a comment after a command but on

the same line. Although you will need to learn Stata's programming features to set up these do-files, doing so is well worth the effort.

First, use `describe` and `summarize` to get useful information about the data you have imported (typically by using `insheet`, `infile`, or `infix`).[13] Consider a version of the `census2a` dataset that has been altered to illustrate data validation:

```
. use http://www.stata-press.com/data/imeus/census2b, clear
(Version of census2a for data validation purposes)
. describe
Contains data from census2b.dta
  obs:            50                          Version of census2a for data
                                                validation purposes
 vars:             5                          23 Sep 2004 15:49
 size:         1,850 (99.9% of memory free)
-------------------------------------------------------------------------------
              storage  display     value
variable name   type   format      label      variable label
-------------------------------------------------------------------------------
state           str14  %14s
region          str7   %9s
pop             float  %9.0g
medage          float  %9.0g
drate           float  %9.0g
-------------------------------------------------------------------------------
Sorted by:
. summarize
    Variable |       Obs        Mean    Std. Dev.       Min        Max
-------------+--------------------------------------------------------
       state |         0
      region |         0
         pop |        49     4392737     4832522        -9   2.37e+07
      medage |        50       35.32    41.25901      24.2        321
       drate |        50       104.3    145.2496        40       1107
```

The log displays the data types of the five variables. The first two are string variables (of maximum length 14 and 7 characters, respectively), whereas other three are `float` variables. These data types appear to be appropriate to the data.

The descriptive statistics reveal several anomalies for the numeric variables. Population data appear to be missing for one state, which is clearly an error. Furthermore, population takes on a negative value for at least one state, indicating some coding errors. We know that the values of U.S. states' populations in recent decades should be greater than several hundred thousand but no more than about 30 million. A median age of 321 would suggest that Ponce de Leon is alive and well. Likewise, the `drate` (death rate) variable has a mean of 104 (per 100,000), so a value of 10 times that number suggests a coding error.

Rather than just firing up the Data Editor and visually scanning for the problems sprinkled through this small illustrative dataset, we are interested in data-validation techniques that we can apply to datasets with thousands of observations. We use `assert`

13. See appendix A for an explanation of these input commands.

to check the validity of these three variables, and in case of failure, we list the offending observations. If all checks are passed, this do-file should run without error:

```
use http://www.stata-press.com/data/imeus/census2b, clear
                                                    // check pop
list if pop < 300000 | pop > 3e7
assert pop < . & pop > 300000 & pop <= 3e7
                                                    // check medage
list if medage <= 20 | medage >= 50
assert  medage > 20  & medage < 50
                                                    // check drate
list if drate < 10   | drate >= 104+145
assert  drate < 10   & drate < 104+145
```

The first `list` command shows that population should not be missing (`< .`, as above), that it should be at least 300,000, and that it should be less than 30 million (3.0×10^7). By reversing the logical conditions in the `list` command, we can assert that all cases have valid values for `pop`.[14] Although the `list` command uses `|`, Stata's "or" operator, the `assert` command uses `&`, Stata's "and" operator, because each condition must be satisfied. Likewise, we `assert` that each state's median age should be between 20 and 50 years. Finally, we assert that the death rate should be at least 10 per 100,000 and less than $\widehat{\mu} + \widehat{\sigma}$ from that variable's descriptive statistics. Let's run the data-validation do-file:

```
. use http://www.stata-press.com/data/imeus/census2b, clear
(Version of census2a for data validation purposes)
. list if pop < 300000 | pop > 3e7
```

	state	region	pop	medage	drate
4.	Arkansas	South	-9	30.6	99
10.	Georgia	South	.	28.7	81
15.	Iowa	N Cntrl	0	30	90

```
. assert pop <. & pop > 300000 & pop <= 3e7
3 contradictions in 50 observations
assertion is false
r(9);
end of do-file
r(9);
```

The do-file fails to run to completion because the first `assert` finds three erroneous values of `pop`. We should now correct these entries and rerun the do-file until it executes without error. This little example could be expanded to a really long do-file that checked each of several hundred variables, and it would exit without error if all assertions are satisfied.

We can use `tabulate` to check the values of string variables in our dataset. In the `census2b` dataset, we will want to use `region` as an identifier variable in later analysis, expecting that each state is classified in one of four U.S. regions.

14. Strictly speaking, we need not apply the `< .` condition, but it is good form to do so since we might not have an upper bound condition.

```
. use http://www.stata-press.com/data/imeus/census2b, clear
(Version of census2a for data validation purposes)
. list state if region==""
```

	state
2.	Alaska
11.	Hawaii

```
. tabulate region
```

region	Freq.	Percent	Cum.
N Cntrl	12	25.00	25.00
NE	9	18.75	43.75
South	16	33.33	77.08
West	11	22.92	100.00
Total	48	100.00	

```
. assert r(N) == 50
assertion is false
r(9);
end of do-file
r(9);
```

The tabulation reveals that only 48 states have `region` defined. We can use one of the items left behind by the `tabulate` command: `r(N)`, the total number of observations tabulated.[15] Here the assertion that we should have 50 defined values of `region` fails, and a list of values where the variable equals string missing (the null string) identifies Alaska and Hawaii as the misclassified entries.

Validating data with `tabulate` can also generate cross tabulations. Consider, for instance, a dataset of medical questionnaire respondents in which we construct a two-way table of `gender` and `NCPreg`, the number of completed pregnancies. Not only should the latter variable have a lower bound of zero and a sensible upper bound, its cross tabulation with `gender=="Male"` should yield only zero values.

We can use `duplicates` to check string variables that should take on unique values. This command can handle much more complex cases—in which a combination of variables must be unique (or a so-called *primary key* in database terminology),[16] but we will apply it to the single variable `state`:

15. Section 2.2.12 discusses the `return list` from r-class commands, such as `tabulate`.
16. As an example: U.S. senators' surnames may not be unique, but the combination of surname and state code almost surely will be unique.

```
. use http://www.stata-press.com/data/imeus/census2b, clear
(Version of census2a for data validation purposes)
. duplicates list state
Duplicates in terms of state
```

obs:	state
16	Kansas
17	Kansas

```
. assert r(sum) == 0
assertion is false
r(9);
end of do-file
r(9);
```

The return item `r(sum)` is set equal to the total number of duplicate observations found (here, 2), so the identification of duplicates implies that you need to correct the dataset. The `duplicates` command could also be applied to numeric variables.

In summary, following sound data-management principles can improve the quality of your data analysis. You should bring the data into Stata as early in the process as possible. Use a well-documented do-file to validate the data, ensuring that variables that should be complete are complete, that unique identifiers are such, and that only sensible values are present in every variable. That do-file should run to completion without error if all data checks are passed. Last, you should not modify the validated and, if necessary, corrected file in later analysis. Subsequent data transformations or merges should create new files rather than overwriting the original contents of the validated file. Following these principles, although time consuming, will ultimately save a good deal of your time and ensure that the data are reproducible and well documented.

Exercises

1. Using the `cigconsumpW` dataset (in long format), merge the `state` variable with that of the *Stata Data Management Reference Manual* dataset `census5` (apply the `uniqusing` option since this dataset is a pure cross section). Compute the averages of `packpc` for subsamples of `median_age` above and below its annual median value (hint: `egen, tabstat`). Does smoking appear to be age related?
2. Using the `cigconsumpW` dataset (in long format), compute 4-year, moving-window averages of `packpc` for each state with `mvsumm`. List the `year`, `packpc`, and `mw4packpc` for California. What do you notice about the moving average relative to the series itself?