JEM174 Stata for Economists
Institute of Economic Studies
Charles University in Prague

Radek Janhuba
janhuba@gmail.com
Spring 2018

Contents

3 Manipulating the Database, Combining Multiple Datasets

3.1 Creating New Variables

In this part of the lecture notes, we will mainly use the subsample of 2009 SOEP data from Germany, which comes from the Kohler & Kreuter (2012) book. Section 5.1 of this book provides more "textbook style" explanation of the topic as well as some additional practical examples.

```
. use data1, clear
(SOEP 2009 (Kohler/Kreuter))
```

3.1.1 A Note on Variable Names

Before we start, note the following about the possibility of variable names in Stata: You can use letters, numbers and underscores, however, names of variables cannot start with a number. Moreover, it is recommended not to start with an underscore.[1]

3.1.2 Generating Using `generate`

The command `generate` is surely one of the most used ones in Stata.[2] Using it is straightforward - we specify the command, followed by the new variable's name and an expression to which the new variable is equal:

```
.  gen newvar = 1

. tab newvar // We can see that all the observations are equal to 1

     newvar |      Freq.     Percent        Cum.
------------+-----------------------------------
          1 |      5,411      100.00      100.00
------------+-----------------------------------
      Total |      5,411      100.00
```

If we want to replace the contents to something else, we can replace it:

```
.  replace newvar = 0
(5,411 real changes made)

. tab newvar // Now all the observations are equal to 0

     newvar |      Freq.     Percent        Cum.
------------+-----------------------------------
          0 |      5,411      100.00      100.00
------------+-----------------------------------
      Total |      5,411      100.00
```

Note that `replace` has exactly the same syntax as `generate`, therefore anything we can do with `gen`, we can do with `replace` - the only difference is that one creates new variables, the other one replaces contents of an existing one. For security reasons, they cannot be used interchangeably (thus, trying `generate` on an existing variable will cause an error):

```
.  gen newvar = 1
variable newvar already defined
r(110);
```

You can create new variables by specifying relations between variables:

```
.  gen age = 2009 - ybirth // generates age
```

[1] The reason for this is that Stata uses names starting with an underscore to store some systemic objects

[2] Note that it can be abbreviated all the way into `g`.

```
. gen age2 = age^2 // age squared

. gen pcinc = hhinc/hhsize // generates per capita inc. as HH inc. div. by HH size
(4 missing values generated)

. list hhinc hhsize pcinc in 1/3

     +--------------------------+
     | hhinc   hhsize     pcinc |
     |--------------------------|
  1. | 22093        2   11046.5 |
  2. | 22093        2   11046.5 |
  3. | 62078        2     31039 |
     +--------------------------+
```

Another way is to use functions that take some variables as an argument: (see `help functions` for a very extensive list of functions available in Stata)

```
.  gen x = rnormal() // generates a random number from N(0,1) distribution

. gen y = abs(x) // generates absolute value of x
```

3.1.3 Creating Dummy Variables

In principle, there are two basic ways how you can create a dummy variable based on the data you have in memory.:

1. generate 0/1 and replace with the other if an expression holds
2. use a direct expression embedded in parenthesis

Suppose we want to have a variable *minor* which equals 1 to people that did not reach age of 18 at the year of the survey (2009). We can use both ways:

```
.  gen minor1 = 0 // equals zero for everyone

. replace minor1 = 1 if age<18 // replaces for 1 for ones born since 1992
(66 real changes made)

.  gen minor2 = (ybirth > 1991) // equals one for minors

. tab minor1 minor2 // We can see they are the same

           |        minor2
    minor1 |         0          1 |     Total
-----------+----------------------+----------
         0 |     5,345          0 |     5,345
         1 |         0         66 |        66
-----------+----------------------+----------
     Total |     5,345         66 |     5,411
```

Note that in this case one does not actually need to use the parenthesis as the expression is the only one entering the calculation.

3.1.4 Generating Using `tabulate`

We have already seen the command `tabulate` which shows us frequencies of observations broken down based on a category. Another extremely useful option is `generate` if you need to create dummy variables based on categories of some variable. You can do this using one line instead of manually creating everything:

```
.  tab state, gen(dum_state)

       State of |
      Residence |      Freq.     Percent        Cum.
----------------+-----------------------------------
         Berlin |        208        3.84        3.84
 Schleswig-Hols. |        166        3.07        6.91
  Hamburg/Bremen |        101        1.87        8.78
    Lower Saxony |        412        7.61       16.39
 N-Rhein-Westfa. |      1,145       21.16       37.55
         Hessen |        355        6.56       44.11
  R-Pfalz,Saarl. |        321        5.93       50.05
Baden-Wuerttemb. |        617       11.40       61.45
         Bavaria |        743       13.73       75.18
  Mecklenburg-V. |        133        2.46       77.64
     Brandenburg |        266        4.92       82.55
   Saxony-Anhalt |        245        4.53       87.08
      Thueringen |        252        4.66       91.74
          Saxony |        447        8.26      100.00
----------------+-----------------------------------
           Total |      5,411      100.00

. describe *state*

              storage   display    value
variable name   type    format     label      variable label
------------------------------------------------------------------------------------------
state           byte    %22.0g     state    * State of Residence
dum_state1      byte    %8.0g                 state==Berlin
dum_state2      byte    %8.0g                 state==Schleswig-Hols.
dum_state3      byte    %8.0g                 state==Hamburg/Bremen
dum_state4      byte    %8.0g                 state==Lower Saxony
dum_state5      byte    %8.0g                 state==N-Rhein-Westfa.
dum_state6      byte    %8.0g                 state==Hessen
dum_state7      byte    %8.0g                 state==R-Pfalz,Saarl.
dum_state8      byte    %8.0g                 state==Baden-Wuerttemb.
dum_state9      byte    %8.0g                 state==Bavaria
dum_state10     byte    %8.0g                 state==Mecklenburg-V.
dum_state11     byte    %8.0g                 state==Brandenburg
dum_state12     byte    %8.0g                 state==Saxony-Anhalt
dum_state13     byte    %8.0g                 state==Thueringen
dum_state14     byte    %8.0g                 state==Saxony

. list state dum_state1 dum_state11 dum_state12 in 8/12
```

```
     +-----------------------------------------------+
     |         state   dum_s~e1   dum_s~11   dum_s~12 |
     |-----------------------------------------------|
  8. | Saxony-Anhalt          0          0          1 |
  9. | Saxony-Anhalt          0          0          1 |
 10. |        Berlin          1          0          0 |
 11. |   Brandenburg          0          1          0 |
 12. |        Berlin          1          0          0 |
     +-----------------------------------------------+
```

3.1.5 Accessing Stored Command Results

One of the most powerful features of Stata is that when you run a command, Stata not only displays output on the results screen, but it also often stores the results in its memory. These results can then be accessed by other functions and used in the analysis/processing. Let's show an example:

```
. sum income // shows descriptive statistics of income

    Variable |        Obs        Mean    Std. Dev.       Min        Max
-------------+---------------------------------------------------------
      income |      4,779     20540.6    37422.49          0     897756
```

In order to see what is stored in the *return* (the name for `r()`), type

```
. return list

scalars:
                  r(N) =  4779
              r(sum_w) =  4779
               r(mean) =  20540.60033479807
                r(Var) =  1400442534.858652
                 r(sd) =  37422.48702129044
                r(min) =  0
                r(max) =  897756
                r(sum) =  98163529
```

We can now see the names of scalars stored in memory along their values. We can use them in calculation or simply display them on the screen:

```
. display r(N) // displays number of observations
4779

. display r(max) // displays the maximum
897756
```

Generally, you can find a list of results stored in memory after a specific command in its help file in section "stored results" (usually towards the bottom).[3]

[3]Hint: stored results are usually easier accessed in the pdf form of the help.

3.1.6 Using Stored Results to Generate Dummy Variables

Suppose now that we need a dummy variable for above average income, which we will call *high_inc*. This can be done in a two step procedure:

```
.  qui sum income // quietly calculates statistics and stores them into r()

. gen high_inc = (income > r(mean)) // generates dummy equal to 1 if above mean

. tab high_inc // look at what was created

   high_inc |      Freq.     Percent        Cum.
------------+-----------------------------------
          0 |      2,918       53.93       53.93
          1 |      2,493       46.07      100.00
------------+-----------------------------------
      Total |      5,411      100.00
```

However, notice that *income* had only 4779 observations, whereas `high_inc` has 5411 (which is the full sample size). This creates a huge error in our data. The reason is that, for technical reasons, Stata treats missing values as large numbers, hence it sets *high_inc* the value of 1 whenever *income* was missing:

```
.  list income high_inc in 1/5

     +-------------------+
     | income   high_inc |
     |-------------------|
  1. |      .          1 |
  2. |      .          1 |
  3. |      0          0 |
  4. |  19955          0 |
  5. |  35498          1 |
     +-------------------+
```

In order to mitigate this issue, we have to use the function `missing()` which returns one if a variable is missing and zero otherwise:

```
.  qui sum income

. gen high_inc2 = (income > r(mean)) if !missing(income)
(632 missing values generated)

. tab high_inc2

  high_inc2 |      Freq.     Percent        Cum.
------------+-----------------------------------
          0 |      2,918       61.06       61.06
          1 |      1,861       38.94      100.00
------------+-----------------------------------
      Total |      4,779      100.00
```

This way, the newly created variable is also missing where the original one was. Remember this in your analysis, otherwise you may get biased results!

3.1.7 Using Constants _n and _N

Suppose now you want to create a number of people interviewed by each interviewer. The variables you are interested are `persnr` and `intnr`. Let's first look at how the data look like:

```
. sort intnr

. list persnr intnr in 1/10, sepby(intnr)

     +------------------+
     |   persnr   intnr |
     |------------------|
  1. |  4035002      10 |
  2. | 10144301      10 |
  3. |  4035003      10 |
  4. |  3934402      10 |
  5. |  4035001      10 |
  6. |  3934401      10 |
     |------------------|
  7. |     8502      18 |
  8. |     8501      18 |
     |------------------|
  9. |  7521603      40 |
 10. |    15001      40 |
     +------------------+
```

Note: the option `sepby` tells Stata to separate list entries after `intnr` changes.

We can now see that we would like to get a variable where interviewer number 10 gets a value of 6, #18 gets 2, etc.

To do this automatically, we will use Stata's inbuilt constants `_n` and `_N`. Type in the following commands:

```
. gen index = _n // generates an overall observation number

. bysort intnr: gen intindex = _n // does the same, but resets for each intnr

. bysort intnr: gen intcount = _N // takes only the largest _n in each intnr
```

Let's now check that the calculation was correct:

```
. list persnr intnr index intindex intcount in 1/10, sepby(intnr)

     +-----------------------------------------------------+
     |   persnr   intnr   index   intindex   intcount |
     |-----------------------------------------------------|
  1. |  4035002      10       1          1          6 |
  2. | 10144301      10       2          2          6 |
  3. |  4035003      10       3          3          6 |
```

```
  4. |  3934402      10       4          4          6 |
  5. |  4035001      10       5          5          6 |
  6. |  3934401      10       6          6          6 |
     |-----------------------------------------------|
  7. |     8502      18       7          1          2 |
  8. |     8501      18       8          2          2 |
     |-----------------------------------------------|
  9. |  7521603      40       9          1         13 |
 10. |    15001      40      10          2         13 |
     +-----------------------------------------------+
```

3.1.8 Accessing i-th Observation in General

In general, an i-th value in Stata can be accessed using the suffix `[i]` behind the variable name. Note that this also means that the last value can be accessed as *var*`[_N]`, the first value as *var*`[1]` etc. Consider this:

```
.  display intnr[1]
10

. display state[_N]
15
```

Example Exercise 3.1

This exercise is based on section 5.1.5. of Kohler & Kreuter (2012). Using the new suffix notation, create

a) A variable containing the total household income
Solution:

```
.  bysort hhnr2009: gen hhearn = sum(income) // calculates rolling sum

. bysort hhnr2009: replace hhearn = hhearn[_N] // replaces with the largest value
(1,658 real changes made)
```

b) A variable containing the social class of head of the household
Solution:

```
.  bysort hhnr2009 (rel2head): gen egphead = egp[1] if rel2head[1]==1
(284 missing values generated)
```

Notes:

1. By putting the variable `rel2head` into parentheses, Stata will sort according to it, however, it will not take it into account when calculating `_n` and `_N` and creating new variables.
2. The `if` construction is only needed because some heads of household may not have been interviewed so the information is not available - Stata will generate missing value in such cases.

3.1.9 `egen` and its Use

Let's get back to the means example above - sometimes we are not happy with simply getting the overall statistics in the dataset, but rather we would like to calculate some statistics for subgroups. As an example, let's now calculate the state-specific high income dummy. In order to do that, we would need to run the above analysis for each state separately as `r()` only remembers the last calculated mean. Luckily, the command `egen` will help us:

```
.  bysort state: egen mean_inc_state = mean(income) // state specific mean income

. gen high_inc_state = (income > mean_inc_state) if !missing(income)
(632 missing values generated)
```

Looking at the data, we can see that different states have different values in the new variable:

```
.  tabstat mean_inc_state, by(state)

Summary for variables: mean_inc_state
     by categories of: state (State of Residence)

           state |      mean
-----------------+----------
          Berlin |  20403.65
 Schleswig-Hols. |  25905.15
  Hamburg/Bremen |  19375.76
    Lower Saxony |  18404.81
 N-Rhein-Westfa. |  21129.48
          Hessen |  27648.83
  R-Pfalz,Saarl. |  19462.12
Baden-Wuerttemb. |  24312.58
         Bavaria |  23314.21
  Mecklenburg-V. |  16076.57
     Brandenburg |  15852.93
   Saxony-Anhalt |  15250.68
      Thueringen |  15732.08
          Saxony |  14010.92
-----------------+----------
           Total |  20510.45
---------------------------
```

Notes to the `egen` command:

1. It's an extension of `gen` command that contains many functions
2. Some of these functions are especially useful if you need to perform some `by` analysis (some, on the other hand, do not allow the `by` prefix)
3. Most of the stuff (or nearly all) that egen does can be replicated by various combinations of `gen` and `replace`.

Some of `egen` options:
Apart from `mean`, which we have already seen, other statistical functions are available:

```
.  bysort state: egen min_inc_state = min(income)

. bysort state: egen max_inc_state = max(income)

. bysort state: egen p95_inc_state = pctile(income), p(95)
```

Let's check that we specified the variables correctly (note that in case all variables in the group have the same value, reporting their mean will report this value)

```
.  tabstat min_inc_state max_inc_state p95_inc_state, stat(mean) by(state)

Summary statistics: mean
  by categories of: state (State of Residence)

           state |  min_in~e  max_in~e  p95_in~e
-----------------+------------------------------
          Berlin |         0    612757     51487
 Schleswig-Hols. |         0    424107     67170
  Hamburg/Bremen |         0     72017     58993
    Lower Saxony |         0    332881     60090
 N-Rhein-Westfa. |         0    324854     65808
          Hessen |         0    897756     74701
  R-Pfalz,Saarl. |         0    210764     55668
Baden-Wuerttemb. |         0    749421   71141.5
         Bavaria |         0    869446     70893
  Mecklenburg-V. |         0     84018     47718
     Brandenburg |         0    109426     51066
   Saxony-Anhalt |         0    110294     53982
       Thueringen |         0    448626     42648
           Saxony |         0     88282     46069
-----------------+------------------------------
            Total |         0    448864  61610.96
--------------------------------------------------

. tabstat income, stat(min max p95) by(state) col(stat)

Summary for variables: income
     by categories of: state (State of Residence)

           state |       min       max       p95
-----------------+------------------------------
          Berlin |         0    612757     51487
 Schleswig-Hols. |         0    424107     67170
  Hamburg/Bremen |         0     72017     58993
    Lower Saxony |         0    332881     60090
 N-Rhein-Westfa. |         0    324854     65808
          Hessen |         0    897756     74701
  R-Pfalz,Saarl. |         0    210764     55668
Baden-Wuerttemb. |         0    749421   71141.5
         Bavaria |         0    869446     70893
  Mecklenburg-V. |         0     84018     47718
```

```
     Brandenburg |         0    109426     51066
   Saxony-Anhalt |         0    110294     53982
      Thueringen |         0    448626     42648
          Saxony |         0     88282     46069
-----------------+------------------------------
           Total |         0    897756     62400
-------------------------------------------------
```

You can also use `egen` to create sums of variables:

```
.  bysort hhnr2009: gen hhinc_sum = sum(income) // cumulative sum (with gen)

. bysort hhnr2009: egen hhinc_tot = total(income) // total sum (not rolling)
```

Note: The first of the two commands creates **cumulative** sum with `generate`. The second creates **total** sum with `egen`. You can see the difference here:

```
.  list hhnr2009 income hhinc_sum hhinc_tot in 1/8

     +-----------------------------------------+
     | hhnr2009   income   hhinc_~m   hhinc_~t |
     |-----------------------------------------|
  1. |       85        .          0          0 |
  2. |       85        .          0          0 |
  3. |      150        0          0      19955 |
  4. |      150    19955      19955      19955 |
  5. |      182        .          0      25514 |
     |-----------------------------------------|
  6. |      182    25514      25514      25514 |
  7. |      265    22694      22694      44258 |
  8. |      265    21564      44258      44258 |
     +-----------------------------------------+
```

Please **avoid** using `sum` option with `egen`. It does the same as `total()` (although undocumented) and can lead to severe confusion.

3.1.10 Creating Categorical Variables with More than Two Categories

In case you need more than one category based on a value of a certain variable, you can use the `recode` command. In this command, you specify a set of rules according to which a new variable should be created / values should be recoded.

Suppose now we want to create age category according to the following rule:

```
1: minors (age 17 or less)
2: adults until 69 years of age
3: seniors (age 70 years or more)
```

This can be done using the following application of `recode`:

```
. recode age (min/17=1 Minor) (18/69=2 Adult) (70/max=3 Senior), generate(agecat)
(5411 differences between age and agecat)
```

Explanation of the command:

- after `recode`, we specify the *varlist* which we want to recode
- after that, we specify a set of rules how we want to recode. These rules can be:

```
a = x  // value "a" will be changed to value "x"
a b = x // values "a" and "b" will be changed to value "x"
a/b = x // values from "a" to "b" will be changed to value "x".
```

  In the last form we can also use `min` and `max` to specify minimum and maximum

- `generate(agecat)` specifies that we want to create a new variable *agecat*. If this option is not specified, recode will replace values in the existing variable.

We can also check that we specified the variable correctly e.g. using `tabstat`:

```
. tabstat age, stat(min max) by(agecat)

Summary for variables: age
     by categories of: agecat (RECODE of age)

agecat |       min       max
-------+--------------------
 Minor |        17        17
 Adult |        18        69
Senior |        70       100
-------+--------------------
 Total |        17       100
---------------------------
```

Example Exercise 3.2.

This example exercise, based on section 5.7 of Kohler & Kreuter (2012), will use a subset of the National Health and Nutrition Examination Study (NHANES) that can be accessed as a web based example dataset:

```
. webuse nhanes2, clear
```

a) Create the variable *men* with value 0 for female observations and 1 for male observations. Label your variable with "Men y/n" and the values of your variables with "no" for 0 and "yes" for 1. **Solution:** Note that there are many ways how one could approach this task. For example, it would be possible to generate the new variable as 0 and then replace with one in appropriate cases, or even copy the new variable and then replace 2 for women with zeros. However, the solution showed below is simpler in the sense that it takes only one line to create the variable.

```
.  gen men = (sex==1) // the most efficient way

. label var men "Men y/n" // label variable

. label define noyes 0 "no" 1 "yes" // define value labels

. label values men noyes // attach value labels

. tab sex men // check whether correct

   1=male, |        Men y/n
  2=female |        no        yes |     Total
-----------+----------------------+----------
      Male |         0      4,915 |     4,915
    Female |     5,436          0 |     5,436
-----------+----------------------+----------
     Total |     5,436      4,915 |    10,351
```

b) Assuming that the formula to calculate the body mass index (BMI) is

$$\text{BMI} = \frac{\text{Weight in kg}}{\text{Height in m}^2}$$

create a fully labeled BMI variable for the NHANES data. Call it `bmi2`.
Solution:

```
.  gen bmi2 = weight / (height/100)^2 // watch out for units (height was in cm)

. corr bmi2 bmi // correlation is 1
(obs=10,351)

             |     bmi2      bmi
-------------+------------------
        bmi2 |   1.0000
         bmi |   1.0000   1.0000

. list bmi2 bmi in 1/3 // the units are correct as well

     +----------------------+
     |     bmi2        bmi |
     |----------------------|
  1. | 20.49569   20.49569 |
  2. | 21.02234   21.02234 |
  3. | 24.97386   24.97386 |
     +----------------------+

. label var bmi2 "BMI"
```

c) Create a fully labeled version of BMI, where the values have been classified according to the following table:

```
Category        BMI
-------------------------
Underweight     < 18.5
Normal weight   18.5-24.9
Overweight      25.0-29.9
Obese   30.0-39.9
Severely obese  > 40
```

Solution: In order to make the code more readable, it will help to change the delimiter first:

```
. #delimit ;
delimiter now ;
. recode bmi                          (min/18.5 = 1 "Underweight")
                        (18.5/25 = 2 "Normal weight")
                        (25/30 = 3 Overweight)
                        (30/40 = 4 Obese)
                        (40/max = 5 "Severely obese")
, gen(bmicat);

(10351 differences between bmi and bmicat)

. #delimit cr
delimiter now cr
. tab bmicat

 RECODE of bmi |
    (Body Mass |
  Index (BMI)) |      Freq.     Percent        Cum.
---------------+-----------------------------------
   Underweight |        329        3.18        3.18
 Normal weight |      5,019       48.49       51.67
    Overweight |      3,396       32.81       84.47
         Obese |      1,465       14.15       98.63
Severely obese |        142        1.37      100.00
---------------+-----------------------------------
         Total |     10,351      100.00
```

Note that you need to use quotation marks to enter labels that include space, or substitute them for underscores. If you try to run it with spaces, Stata will report an error. If you use them with ones that do not need quotation marks, it works as well, as the example above shows.

d) Create a fully labeled version of BMI, where the values of BMI have been classified into four groups with approximately the same number of observations. Label these groups in the following way:

```
1: Low weight
2: Medium-low weight
3: Medium-high weight
4: High weight
```

Solution: We will first show a "semi-manual" solution where we first create cutoff values and then afterwards we define the variable based on these cutoff values:

```
. egen c1 = pctile(bmi), p(25) // 25th percentile

. egen c2 = pctile(bmi), p(50) // 50th

. egen c3 = pctile(bmi), p(75) // 75th

. gen bmicat2 = 1 + (bmi>c1) + (bmi>c2) + (bmi>c3)
```

Notice here how the logical expressions in parentheses can be combined in an arithmetic expression to create the variable in need. Note also that this is only one of multiple ways one can use. Another option would be

```
. gen bmicat2alt = (bmi<=c1) + 2*(bmi>c1 & bmi<=c2) + 3*(bmi>c2 & bmi<=c3) + ///
                                4*(bmi>c3)
```

which results in exactly the same variable being created.

In order to keep the dataset clean of unnecessary variables, it may be a good idea to drop the cutoffs after using this solution:

```
. drop c1 c2 c3
```

The alternative, much simpler option, is to use the `cut` function of `egen`:

```
. egen bmicat2cut = cut(bmi), group(4) // generates 0 to 3

. replace bmicat2cut = bmicat2cut + 1 // replaces to 1 to 4
(10,351 real changes made)
```

In this case, all the steps above are done automatically and the created variable is the same. Notes:

- The method using `cut` function of `egen` is definitely easier to use, however, unfortunately it does not allow `by` to be utilized. If we therefore asked for e.g. state specific categories, we would need to use the first method.
- In any case, we still need to label the variable. This is done by

```
. label define bmicat2 1 "Low weight" 2 "Medium-low weight" ///
                                        3 "Medium-high weight" 4 "High weight"

. label values bmicat2 bmicat2alt bmicat2cut bmicat2 // attach label

. list bmicat2 bmicat2alt bmicat2cut in 1/4 // labeled now
```

```
     +-------------------------------------------------------------+
     |            bmicat2           bmicat2alt           bmicat2cut |
     |-------------------------------------------------------------|
  1. |         Low weight           Low weight           Low weight |
  2. |         Low weight           Low weight           Low weight |
  3. | Medium-high weight   Medium-high weight   Medium-high weight |
  4. |        High weight          High weight          High weight |
     +-------------------------------------------------------------+
```

e) Create a fully labeled version of BMI, where the values of BMI have been classified into three groups defined as follows:

- Group 1 contains observations with a BMI lower or equal to one standard deviation below the gender-specific average.
- Group 2 contains observations with values higher than group 1 but lower than one standard deviation above the gender-specific average.
- Group 3 contains observations with values above group 2.

Solution: Looking at `help egen`, we can see that there is a function `sd()` that will help us with this task. Similarly to above, we will define cutoffs and then create the new variable. However, as `egen` does not allow for combinations of functions, we get an error when we write

```
. bysort sex: egen c1 = mean(bmi) - sd(bmi)
varlist not allowed
r(101);
```

Therefore, we have to take a little different approach:

```
. bysort sex: egen m = mean(bmi) // generates gender specific mean

. bysort sex: egen sd = sd(bmi) // generates gender specific time
```

Now, here are two alternative options which both create exactly the same and it is only up to you which of these you use. While option *a* is more straightforward, it also requires longer typing. To the contrary, option *b* is more efficient, but is little more abstract.

```
. gen bmicat3a = (bmi <= m - sd) + 2*((bmi > m - sd) & (bmi < m + sd) ) ///
                                 + 3* (bmi >= m + sd)

. gen bmicat3b = 2 - (bmi <= m - sd) + (bmi >= m + sd)

. bysort bmicat3a bmicat3b: tabstat bmi, stat(min max) by(sex)

-------------------------------------------------------------------------------
-> bmicat3a = 1, bmicat3b = 1

Summary for variables: bmi
     by categories of: sex (1=male, 2=female)
```

```
   sex |       min       max
-------+--------------------
  Male |   12.3856  21.48568
Female |   14.1351  19.95063
-------+--------------------
 Total |   12.3856  21.48568
----------------------------

-------------------------------------------------------------------------------
-> bmicat3a = 2, bmicat3b = 2

Summary for variables: bmi
     by categories of: sex (1=male, 2=female)

   sex |       min       max
-------+--------------------
  Male |  21.48633  29.53256
Female |  19.96479  31.16088
-------+--------------------
 Total |  19.96479  31.16088
----------------------------

-------------------------------------------------------------------------------
-> bmicat3a = 3, bmicat3b = 3

Summary for variables: bmi
     by categories of: sex (1=male, 2=female)

   sex |       min       max
-------+--------------------
  Male |  29.53649  53.04031
Female |  31.16777   61.1297
-------+--------------------
 Total |  29.53649   61.1297
----------------------------
```

Note: Labeling of this variable is not shown here, we have already seen it multiple times.

3.2 Processing Strings

This section will show us how to work with string variables in Stata. Before we start, let's open another dataset coming from Kohler & Kreuter (2012) textbook. The dataset uses information on all German politicians who were members of the parliament (Bundestag) between 1949 and 1998.

```
.  use mdb, clear
(MoP, Germany 1949-1998)

. describe

Contains data from mdb.dta
```

```
  obs:         7,918                          MoP, Germany 1949-1998
 vars:            14                          16 Jul 2012 11:39
 size:       934,324                          (_dta has notes)
-------------------------------------------------------------------------------
              storage   display    value
variable name   type    format     label      variable label
-------------------------------------------------------------------------------
index           int     %8.0g                 Index-Number for Parlamentarian
name            str63   %63s                  Name of Parlamentarian
party           str10   %10s                  Fraction-Membership
period          str2    %9s                   Legislative Period
pstart          int     %d                    Start of Legislative Period
pend            int     %d                    End of Legislative Period
constituency    str4    %9s                   Voted in Constituency/Country Party Ticket
birthyear       int     %9.0g                 Year of Birth
birthmonth      byte    %9.0g                 Month of Birth
birthday        byte    %9.0g                 Day of Birth
deathdate       int     %d                    Date of Death
begindate       str15   %15s                  Begin of Episode
enddate         str11   %11s                  End of Episode
endtyp          byte    %25.0g     endtyp     Reason for Leaving the Parliament
-------------------------------------------------------------------------------
Sorted by: pstart  name
```

3.2.1 Simple String Addition

Suppose that we now want to generate a "TV subtitle" where each politician will have their name followed by their political party in parentheses. Looking at what we have,

```
. list name party in 1/3

     +-------------------------------+
     |                 name   party |
     |-------------------------------|
  1. | Adenauer, Dr. Konrad     CDU |
  2. |         Agatz, Willi     KPD |
  3. |        Ahrens, Adolf      DP |
     +-------------------------------+
```

we would like to create *Agatz, Willi (KPD)* in the second line etc.:

```
. gen tvsub = name + " (" + party + ")"
```

Notice the space in front of the first parenthesis - it is necessary to put it there as the command adds everything exactly as you tell it

```
. list name party tvsub in 1/3

     +-----------------------------------------------------------+
     |                 name   party                        tvsub |
```

```
     |-------------------------------------------------------|
  1. | Adenauer, Dr. Konrad     CDU   Adenauer, Dr. Konrad (CDU) |
  2. |          Agatz, Willi     KPD           Agatz, Willi (KPD) |
  3. |         Ahrens, Adolf      DP           Ahrens, Adolf (DP) |
     +-------------------------------------------------------+
```

3.2.2 Using Numbers in String Expressions

Suppose now that we want to generate *Name, yyyy* where *yyyy* is the year of birth. Notice that if you write

```
. gen nameborn = name + birthyear
type mismatch
r(109);
```

you will get a "type mismatch" error. This is because `generate` is not able to combine strings with numerical variables by default. In order to combine such variables, we need to use a function `string()`:

```
. gen nameborn = name + ", " + string(birthyear)
```

3.2.3 Controlling Cases

Suppose now you want to create a dummy variable for politicians from CDU:

```
. gen cdu = (party=="CDU")

. tab cdu // there are observations matching the party

        cdu |      Freq.     Percent        Cum.
------------+-----------------------------------
          0 |      4,331       54.70       54.70
          1 |      3,587       45.30      100.00
------------+-----------------------------------
      Total |      7,918      100.00
```

Even though we did not control for lower or upper case in the name, the procedure worked. In case of SPD, if we write the following, we will not be so lucky:

```
. gen spd = (party=="spd")

. tab spd // we will have zero observations equal to one

        spd |      Freq.     Percent        Cum.
------------+-----------------------------------
          0 |      7,918      100.00      100.00
------------+-----------------------------------
      Total |      7,918      100.00
```

The reason is the case sensitivity of Stata. If we write

```
.  gen spd2 = (lower(party)=="spd")

. tab spd2 // now there will be observations

       spd2 |      Freq.     Percent        Cum.
------------+-----------------------------------
          0 |      4,995       63.08       63.08
          1 |      2,923       36.92      100.00
------------+-----------------------------------
      Total |      7,918      100.00
```

we will get what we need, as function `lower()` takes the argument and makes it lowercase, while its counterpart, `upper()` creates an uppercase string:

```
.  display lower("Stata")
stata

. display upper("Stata")
STATA
```

It is recommended to use one of these functions whenever one needs to perform some logical operations with strings that had been manually typeset.

3.2.4 String Positions

Function `strpos(`*s1*`,`*s2*`)` checks whether a string *s2* is part of another string *s1*, and returns the position where *s2* is first found in *s1*. For example, you can find where the first letter "a" is in Stata:

```
.  display strpos("Stata","a")
3
```

Recalling the names of politicians that we have in the dataset,

```
.  list name in 1/3

     +----------------------+
     |                 name |
     |----------------------|
  1. | Adenauer, Dr. Konrad |
  2. |         Agatz, Willi |
  3. |        Ahrens, Adolf |
     +----------------------+
```

we can see that some of them have a doctorate degree, indicated by having "Dr." listed in their name. If we want a dummy for being a doctor, we can create it as

```
.  gen doctor = (strpos(name,"Dr.")>0)
```

Note that this way utilizes the fact that Stata returns position as zero in case *s2* is not a part of *s1*).

3.2.5 Extracting Substrings

Function `substr(s,n1,n2)` returns the substring of *s*, starting at column *n1*, for a length of *n2*:

```
. display substr("Stata for Economists 2018",1,5)
Stata
```

If $n1 < 0$, *n1* is interpreted as distance from the end of the string:

```
. display substr("Stata for Economists 2018",-19,14)
for Economists
```

If $n2 = .$ (missing), the remaining portion of the string is returned:

```
. display substr("Stata for Economists 2018",-4,.)
2018
```

Suppose now we want to break politicians' names into their family names and their given names. In order to do that, we first need to find out where the comma is in each name:

```
. gen comma = (strpos(name,","))
```

Now, we can create their family name as

```
. gen famname = substr(name,1,comma-1)

. list name famname in 1/3 // we now have a variable with family names

     +---------------------------------+
     |                 name    famname |
     |---------------------------------|
  1. | Adenauer, Dr. Konrad   Adenauer |
  2. |         Agatz, Willi      Agatz |
  3. |        Ahrens, Adolf     Ahrens |
     +---------------------------------+
```

3.2.6 Replacing Substrings

Suppose we want to get first names as well:

```
. gen firstname = substr(name,comma+2,.) // start after the space after comma

. list firstname in 1/3

     +------------+
     |  firstname |
     |------------|
  1. | Dr. Konrad |
  2. |      Willi |
  3. |      Adolf |
     +------------+
```

We can see that we now have variable with family names. However, we would like to eliminate "Dr. " in the beginning. This is where the new function comes in: `subinstr(s1,s2,s3,n)` returns *s1*, where the first *n* occurrences of *s2* in *s1* have been replaced with *s3*. Some demonstrative examples are:

```
. display subinstr("Stata for Economists 2018","o","O",2)
Stata fOr EcOnomists 2018
```

If n is missing (.), all occurences are replaced:

```
. display subinstr("Stata for Economists 2018","o","O",.)
Stata fOr EcOnOmists 2018
```

In order to eliminate "Dr. " from the first name, we can write

```
. replace firstname = subinstr(firstname,"Dr. ","",.) // replace "Dr. " with ""
(2,184 real changes made)

. list firstname in 1/3 // we now have a variable with first names

     +----------+
     | firstn~e |
     |----------|
  1. |   Konrad |
  2. |    Willi |
  3. |    Adolf |
     +----------+
```

Notes:

- We need to use `replace` instead of `gen`, as we have already created the *firstname* variable.
- We do need to put double quotes as an "empty string" into the third argument.

3.2.7 A "text-to-columns"

Sometimes you need to split a string based on a certain delimiter. Let's now try to split up the "name" into the first and last name in a bit different way:

```
. list name in 1/3

     +----------------------+
     |                 name |
     |----------------------|
  1. | Adenauer, Dr. Konrad |
  2. |         Agatz, Willi |
  3. |        Ahrens, Adolf |
     +----------------------+

. split name, parse(", ") // Break into surname and "doctorfirstname"
variables created as string:
name1  name2  name3
```

Notice that, unexpectedly, we obtained three new variables (we would expect only two). Let's look at where is the issue:

```
. list name if !missing(name3)

     +---------------------+
     |                name |
     |---------------------|
4873. | Keller, Peter, M.A. |
5433. | Keller, Peter, M.A. |
6068. | Keller, Peter, M.A. |
6796. | Keller, Peter, M.A. |
7492. | Keller, Peter, M.A. |
     +---------------------+
```

The issue therefore is that one person had an M.A. degree, causing it to appear as a third name.

3.3 Processing Dates

By default, Stata treats dates as a number of elapsed days since Jan 1, 1960. However, the dates can be stored in different ways:

```
. list name pstart pend in 1/3 // The proper way how dates should be stored

   +-----------------------------------------------+
   |                 name      pstart        pend |
   |-----------------------------------------------|
 1. | Adenauer, Dr. Konrad   07sep1949   07sep1953 |
 2. |         Agatz, Willi   07sep1949   07sep1953 |
 3. |        Ahrens, Adolf   07sep1949   07sep1953 |
   +-----------------------------------------------+

. list name begindate enddate in 1/3 // Dates stored as strings

   +-------------------------------------------------------+
   |                 name         begindate       enddate |
   |-------------------------------------------------------|
 1. | Adenauer, Dr. Konrad   07 September 49   Sep 07 1953 |
 2. |         Agatz, Willi   07 September 49   Sep 07 1953 |
 3. |        Ahrens, Adolf   07 September 49   Sep 07 1953 |
   +-------------------------------------------------------+

. list name birth* in 1/3 // Dates stored in three separate variables

   +-----------------------------------------------------------+
   |                 name   birthy~r   birthm~h   birthday |
   |-----------------------------------------------------------|
 1. | Adenauer, Dr. Konrad       1876          1          5 |
 2. |         Agatz, Willi       1904          6         10 |
 3. |        Ahrens, Adolf       1879          9         17 |
   +-----------------------------------------------------------+
```

3.3.1 Creating Dates from Numbers

Let's now create the `birthdate` variable which will contain the numeric value of the date:

```
. gen birthdate = mdy(birthmonth,birthday,birthyear)
```

In this application, we used the function `mdy` which takes 3 arguments (month, day and year) and returns a number associated with the particular date.

```
. list name birth* in 1/3
```

```
     +-------------------------------------------------------------------+
     |                 name   birthy~r   birthm~h   birthday   birthd~e |
     |-------------------------------------------------------------------|
  1. | Adenauer, Dr. Konrad       1876          1          5     -30676 |
  2. |          Agatz, Willi      1904          6         10     -20293 |
  3. |         Ahrens, Adolf      1879          9         17     -29325 |
     +-------------------------------------------------------------------+
```

Notice that the variable is really stored as numeric, moreover, these numbers are negative as they are before 1st January 1960. However, these values are not informative about what they really represent. We can fix that by attaching a proper format to it:

```
. format birthdate %td

. list name birth* in 1/3
```

```
     +-------------------------------------------------------------------+
     |                 name   birthy~r   birthm~h   birthday   birthdate |
     |-------------------------------------------------------------------|
  1. | Adenauer, Dr. Konrad       1876          1          5   05jan1876 |
  2. |          Agatz, Willi      1904          6         10   10jun1904 |
  3. |         Ahrens, Adolf      1879          9         17   17sep1879 |
     +-------------------------------------------------------------------+
```

The `%td` is the default display for dates in Stata. However, there are many more possible formats available (see `help datetime_display_formats` for an overview). For example:

```
. format birthdate %tdM_d,_CY

. list name birth* in 1/3
```

```
     +-------------------------------------------------------------------------+
     |                 name   birthy~r   birthm~h   birthday           birthdate |
     |-------------------------------------------------------------------------|
  1. | Adenauer, Dr. Konrad       1876          1          5      January 5, 1876 |
  2. |          Agatz, Willi      1904          6         10        June 10, 1904 |
  3. |         Ahrens, Adolf      1879          9         17   September 17, 1879 |
     +-------------------------------------------------------------------------+
```

3.3.2 Creating Dates from Strings

If we have a date stored in a string variable (in this case *begindate* and *enddate*), we can use the function `date()`. This function takes a date stored in a string in one of multiple formats and converts it to a Stata date (with just a little help which specifies the order of day, month, and year). Take the date of October 28:

```
.  display date("28.10.2015","DMY")
20389
```

or better (notice how we can specify the format of the displayed string)

```
.  display %td date("28.10.2015","DMY")
28oct2015

. display %td date("10/28/2015","MDY")
28oct2015

. display %td date("2015: Oct 28","YMD")
28oct2015
```

Note that if we want to use two digit instead of four digit years, we need to either specify a century or the maximum year we allow to be created:

```
.  display %td date("10/28/15","MDY")  // returns missing
        .

. display %td date("10/28/15","MD20Y")
28oct2015

. display %td date("10/28/15","MDY",2020)
28oct2015

. display %td date("10/28/15","MDY",2000) // returns 1915
28oct1915
```

3.3.3 Extracting Info from Dates

If you have a date stored properly in a numeric format, you can use functions `month()`, `day()` and `year()` to extract parts of the dates. Suppose we want to create a dummy for all politicians that were born on the day of the beginning of the course:

```
.  gen borncourse = (day(birthdate)==12) & (month(birthdate)==2)

. tab name if borncourse

                Name of Parlamentarian |      Freq.     Percent        Cum.
---------------------------------------+-----------------------------------
                     Amende, Andreas |          1        4.00        4.00
                   Boehme, Dr. Ulrich |          3       12.00       16.00
```

```
          Bruehler, Dr. Ernst-Christoph |          2        8.00       24.00
                      Eppelmann, Rainer |          2        8.00       32.00
                       Giulini, Dr. Udo |          2        8.00       40.00
                       Heimann, Gerhard |          2        8.00       48.00
                  Hoffmann, Hans-Joachim |          3       12.00       60.00
                            Hoppe, Anton |          1        4.00       64.00
                     Janssen, Jann-Peter |          1        4.00       68.00
                         Junglas, Johann |          1        4.00       72.00
                     Lamers, Dr. Karl A. |          1        4.00       76.00
                  Muehlhan, Dr. Bernhard |          1        4.00       80.00
                         Schell, Manfred |          1        4.00       84.00
                         Schmidt, Thomas |          1        4.00       88.00
                       Vahlberg, Juergen |          3       12.00      100.00
-------------------------------------------+-----------------------------------
                                    Total |         25      100.00
```

Alternatively, in case you would like to do it automatically for each day you run a code, you can do so using the following code:

```
.  scalar today = date(c(current_date),"DMY")

. gen borntoday2 = (day(birthdate)==day(today)) & (month(birthdate)==month(today))
```

See `help scalar` or `help creturn` for an explanation of these commands. We will get back to scalars later on, however, we will leave creturn to interested readers only.

Example Exercise 3.3

Find out who was the youngest politician ever to enter the Bundestag.
Solution: Looking at the data, we will need "birthdate" (we already calculated it) and "begindate". The first one is already in the proper format, but the second one is a string.

```
.  gen start = date(begindate,"DM19Y") // convert it to numeric values

. gen startage = start-birthdate // calculate their age in days when started

. gen startagey = startage / 365.25 // .25 because of leap years

. sort startagey // sort by the calculated variable

. list name startagey in 1 // list the name of the "winner"

     +--------------------------------+
     |                name   starta~y |
     |--------------------------------|
  1. | Berninger, Matthias    23.7755 |
     +--------------------------------+
```

Notes:

- the first three lines may be combined as

```
. gen startagey2 = (date(begindate,"DMY",2000) - birthdate)/365.25

. list name startagey2 in 1

     +--------------------------------+
     |                name   starta~2 |
     |--------------------------------|
  1. | Berninger, Matthias    23.7755 |
     +--------------------------------+
```

- The above solution only works thanks to the fact that the solution is unique - in case there would be two or more politicians tied for the first place, we would get an incorrect answer. The following code is a more general solution, revealing all that satisfy the conditions for the solution:

```
. qui sum startagey // quietly sums the variable in question

. tab name startagey if startagey==r(min) // show the "winner(s)"

             Name of | startagey
      Parlamentarian |   23.7755 |     Total
---------------------+-----------+----------
  Berninger, Matthias |         1 |         1
---------------------+-----------+----------
               Total |         1 |         1
```

3.4 Dealing with Duplicates

Often times you will be interested if you have duplicate observations in your data, with the reason being e.g. a need to reshape the dataset from wide to long form (see the next section), which you will not be able to perform if there are duplicate observations. In order to report, tag, or drop duplicate observations, command `duplicates` can be used in association with several subcommands.

In order to show what the command can do, we will use an example dataset provided by Stata. Let's now open and look at the dataset:

```
. webuse dupxmpl, clear

. list in 1/3

     +------------+
     | id   x   y |
     |------------|
  1. |  1   0   1 |
  2. |  2   0   1 |
  3. |  3   0   1 |
     +------------+

. qui sum id // sum quietly -> does not show output

. disp r(max)
200
```

```
. disp _N
202
```

This dataset contains an id of a unit and then x and y coordinates. However, as we can see the maximum id is 200 and the dataset contains 202 observations. Chances are that this was hand-input and something was typed in twice by accident. In order to check whether there are any duplicates, we can use the `report` subcommand:

```
.  duplicates report

Duplicates in terms of all variables

--------------------------------------
   copies | observations       surplus
----------+---------------------------
        1 |          198             0
        2 |            4             2
--------------------------------------
```

We can see that there are 4 observations that are duplicate in 2 copies, meaning that likely two ids were input twice. To see which these were, we can use `list`:

```
.  duplicates list

Duplicates in terms of all variables

  +-----------------------------+
  | group:   obs:    id   x   y |
  |-----------------------------|
  |      1     42    42   0   2 |
  |      1     43    42   0   2 |
  |      2    145   144   4   4 |
  |      2    146   144   4   4 |
  +-----------------------------+
```

This subcommand lists all duplicate observations, which may sometimes be too exhaustive (imagine having millions of observations where there are hundreds of duplicates). To go around, we can use the `example` subcommand which always shows only the first instance of a given duplicate:

```
.  duplicates examples

Duplicates in terms of all variables

  +-----------------------------------+
  | group:   #   e.g. obs    id   x   y |
  |-----------------------------------|
  |      1   2         42    42   0   2 |
  |      2   2        145   144   4   4 |
  +-----------------------------------+
```

If we want to create a variable that identifies the duplicates, we can use the `tag` subcommand. This generates a variable including the number of duplicates for each observation in the dataset:

```
.  duplicates tag, gen(dupl)

Duplicates in terms of all variables

. list in 41/45

     +------------------+
     | id   x   y   dupl |
     |------------------|
 41. | 41   1   1      0 |
 42. | 42   0   2      1 |
 43. | 42   0   2      1 |
 44. | 43   0   1      0 |
 45. | 44   3   1      0 |
     +------------------+
```

Last but not least, we can drop duplicate observations using the `drop` subcommand:

```
.  duplicates drop

Duplicates in terms of all variables

(2 observations deleted)

. list in 41/45

     +------------------+
     | id   x   y   dupl |
     |------------------|
 41. | 41   1   1      0 |
 42. | 42   0   2      1 |
 43. | 43   0   1      0 |
 44. | 44   3   1      0 |
 45. | 45   0   1      0 |
     +------------------+
```

Notes to duplicates

- Do **not** write `duplicates tag` and then `drop if tag>0`. This way **all** of the duplicate observations would get dropped, not just the ones that are actually redundant.
- You can also identify the duplicates based on not all variables, but only some selected varlist, which you indicate by typing the varlist after the subcommand. Let's indicate the difference on the *auto* example dataset:

```
.  sysuse auto, clear
(1978 Automobile Data)
```

```
. duplicates examples // no duplicates in terms of all variables

Duplicates in terms of all variables

(0 observations are duplicates)

. duplicates examples weight length // some duplicates in terms of "size"

Duplicates in terms of weight length

  +---------------------------------------+
  | group:   #   e.g. obs   weight   length |
  |---------------------------------------|
  |      1   2         44    2,200      165 |
  |      2   2         21    3,600      206 |
  +---------------------------------------+

. duplicates tag weight length, gen(dupl_size) // create a tag

Duplicates in terms of weight length

. list make mpg weight length if dupl_size > 0 // list duplicates with make

     +---------------------------------------+
     | make             mpg   weight   length |
     |---------------------------------------|
 21. | Dodge Diplomat    18    3,600      206 |
 22. | Dodge Magnum      16    3,600      206 |
 44. | Plym. Horizon     25    2,200      165 |
 68. | Toyota Corolla    31    2,200      165 |
     +---------------------------------------+
```

3.5 (Re)shaping the Dataset: Long vs. Wide

This section will be especially useful if you have a panel dataset, but it can be applied to non-panel sets as well. Imagine you have the following dataset, which can be transformed between the two following forms:

```
long
+------------+                  wide
| i  j  stub |                 +----------------+
|------------|                 | i  stub1 stub2 |
| 1  1   4.1 |     reshape     |----------------|
| 1  2   4.5 |  <------------> | 1    4.1   4.5 |
| 2  1   3.3 |                 | 2    3.3   3.0 |
| 2  2   3.0 |                 +----------------+
+------------+
```

To go from left to right, write

```
reshape wide stub, i(i) j(j) // here j is an existing variable
```

To go from wide to long (right to left):

```
reshape long stub, i(i) j(j) // here j is a new variable
```

Let's now look at some examples of how this can be applied in actual data:

```
. webuse reshape1, clear

. list

     +-----------------------------------------------------+
     | id   sex   inc80   inc81   inc82   ue80   ue81   ue82 |
     |-----------------------------------------------------|
  1. |  1     0    5000    5500    6000      0      1      0 |
  2. |  2     1    2000    2200    3300      1      0      0 |
  3. |  3     0    3000    2000    1000      0      0      1 |
     +-----------------------------------------------------+
```

We can see that we have variables *inc* and *ue* for years 1980 to 1982 stored in a wide form. Let's convert them to the long form:

```
. reshape long inc ue, i(id) j(year) // id identifies, j is created
(note: j = 80 81 82)

Data                               wide   ->   long
-----------------------------------------------------------------------------
Number of obs.                        3   ->       9
Number of variables                   8   ->       5
j variable (3 values)                     ->   year
xij variables:
                     inc80 inc81 inc82   ->   inc
                        ue80 ue81 ue82   ->   ue
-----------------------------------------------------------------------------

. list, sepby(id) // separate by id for nicer readability

     +-----------------------------+
     | id   year   sex    inc   ue |
     |-----------------------------|
  1. |  1     80     0   5000    0 |
  2. |  1     81     0   5500    1 |
  3. |  1     82     0   6000    0 |
     |-----------------------------|
  4. |  2     80     1   2000    1 |
  5. |  2     81     1   2200    0 |
  6. |  2     82     1   3300    0 |
     |-----------------------------|
  7. |  3     80     0   3000    0 |
  8. |  3     81     0   2000    0 |
  9. |  3     82     0   1000    1 |
     +-----------------------------+
```

We can see that we had 3 observations and now we have 9, but at the same time we had 8 variables and now we only have 5. In order to get back to the original form of the dataset, we can write

```
.  reshape wide inc ue, i(id) j(year)
(note: j = 80 81 82)

Data                               long   ->   wide
-----------------------------------------------------------------------------
Number of obs.                        9   ->       3
Number of variables                   5   ->       8
j variable (3 values)              year   ->   (dropped)
xij variables:
                                    inc   ->   inc80 inc81 inc82
                                     ue   ->   ue80 ue81 ue82
-----------------------------------------------------------------------------

. list

     +-------------------------------------------------------+
     | id   inc80   ue80   inc81   ue81   inc82   ue82   sex |
     |-------------------------------------------------------|
  1. |  1    5000      0    5500      1    6000      0     0 |
  2. |  2    2000      1    2200      0    3300      0     1 |
  3. |  3    3000      0    2000      0    1000      1     0 |
     +-------------------------------------------------------+
```

We can see that the dataset is again in the original order and that *year* was dropped.

Note: see the pdf documentation of `reshape` to get excellent examples on mistakes that can arise while using `reshape` and how to avoid them.

3.6 Appending Observations to Existing Dataset

The command `append` lets you combine multiple Stata data files to each other. In order to illustrate the functionality, let's now load and save a web-based example datasets called *even* and *odd*.

```
.  webuse even, clear
(6th through 8th even numbers)

. list // we have only even numbers now

     +---------------+
     | number   even |
     |---------------|
  1. |      6     12 |
  2. |      7     14 |
  3. |      8     16 |
     +---------------+

. save even, replace
file even.dta saved
```

```
. webuse odd, clear
(First five odd numbers)

. list // we have only odd data now

     +--------------+
     | number   odd |
     |--------------|
  1. |      1     1 |
  2. |      2     3 |
  3. |      3     5 |
  4. |      4     7 |
  5. |      5     9 |
     +--------------+
```

at this point, we do not have to save the *odd* dataset as we have it loaded in memory. We can, however, add the observations from the "even" dataset to the odd dataset:

```
.  append using even

. list // We now have a combined dataset out of the two files.

     +---------------------+
     | number   odd   even |
     |---------------------|
  1. |      1     1      . |
  2. |      2     3      . |
  3. |      3     5      . |
  4. |      4     7      . |
  5. |      5     9      . |
     |---------------------|
  6. |      6     .     12 |
  7. |      7     .     14 |
  8. |      8     .     16 |
     +---------------------+
```

There are two more functionalities of append that we will cover:

1. you can specify more than one file to append
2. you can use the option `generate` to create a name of variable which will indicate which dataset the observation came from.

The following example illustrates these two functionalities:

```
.  webuse capop, clear // loads example counties and populations from California

. list // we have three observations

     +-----------------------+
     |      county       pop |
```

```
     |----------------------|
  1. | Los Angeles   9878554 |
  2. |      Orange   2997033 |
  3. |     Ventura    798364 |
     +----------------------+

. save capop, replace // save
file capop.dta saved

. webuse ilpop, clear // Illinois

. list

     +------------------+
     | county       pop |
     |------------------|
  1. |   Cook   5285107 |
  2. | DeKalb    103729 |
  3. |   Will    673586 |
     +------------------+

. save ilpop, replace
file ilpop.dta saved

. webuse txpop, clear // Texas

. list

     +-------------------+
     |  county       pop |
     |-------------------|
  1. |  Brazos    152415 |
  2. | Johnson    149797 |
  3. |  Harris   4011475 |
     +-------------------+

. save txpop, replace
file txpop.dta saved

. append using ilpop capop, gen(state) // Note we currently have Texas in memory
(note: variable county was str7, now str11 to accommodate using data's values)

. list, sepby(state) // state generated, but not very informative

     +-------------------------------+
     |      county       pop   state |
     |-------------------------------|
  1. |      Brazos    152415       0 |
  2. |     Johnson    149797       0 |
  3. |      Harris   4011475       0 |
     |-------------------------------|
```

```
  4. |        Cook   5285107       1 |
  5. |      DeKalb    103729       1 |
  6. |        Will    673586       1 |
     |-------------------------------|
  7. | Los Angeles   9878554       2 |
  8. |      Orange   2997033       2 |
  9. |     Ventura    798364       2 |
     +-------------------------------+

. la def state 0 "TX" 1 "IL" 2 "CA" // define label

. la val state state // attach label

. list, sepby(state) // state is now informative

     +-------------------------------+
     |      county       pop   state |
     |-------------------------------|
  1. |      Brazos    152415      TX |
  2. |     Johnson    149797      TX |
  3. |      Harris   4011475      TX |
     |-------------------------------|
  4. |        Cook   5285107      IL |
  5. |      DeKalb    103729      IL |
  6. |        Will    673586      IL |
     |-------------------------------|
  7. | Los Angeles   9878554      CA |
  8. |      Orange   2997033      CA |
  9. |     Ventura    798364      CA |
     +-------------------------------+
```

Note that you can also append multiple files starting with the empty dataset. In such case the only difference will be that the generated value will start with 1 (there will be no group with 0):

```
.  clear // clears dataset

. list // confirms we don't have any data

. append using txpop ilpop capop, generate(state) // append + gen state id
(note: variable county was str7, now str11 to accommodate using data's values)

. list, sepby(state) // state2 ranges from 1 to 3 rather than 0 to 2

     +-------------------------------+
     | state        county       pop |
     |-------------------------------|
  1. |     1        Brazos    152415 |
  2. |     1       Johnson    149797 |
  3. |     1        Harris   4011475 |
     |-------------------------------|
  4. |     2          Cook   5285107 |
```

```
  5. |     2        DeKalb    103729 |
  6. |     2          Will    673586 |
     |-------------------------------|
  7. |     3   Los Angeles   9878554 |
  8. |     3        Orange   2997033 |
  9. |     3       Ventura    798364 |
     +-------------------------------+

. la def state2 1 "TX" 2 "IL" 3 "CA" // numbers increased by 1

. label values state state2 // attach label

. list, sepby(state) // Same result

     +-------------------------------+
     | state        county       pop |
     |-------------------------------|
  1. |    TX        Brazos    152415 |
  2. |    TX       Johnson    149797 |
  3. |    TX        Harris   4011475 |
     |-------------------------------|
  4. |    IL          Cook   5285107 |
  5. |    IL        DeKalb    103729 |
  6. |    IL          Will    673586 |
     |-------------------------------|
  7. |    CA   Los Angeles   9878554 |
  8. |    CA        Orange   2997033 |
  9. |    CA       Ventura    798364 |
     +-------------------------------+
```

Notes:

- `append` should be used **only** in cases where you need to add some observations to an existing dataset (like an update when you receive data for an additional year). In case you need to add some variables, see the command `merge` below.
- In order to for `append` to add observations to a current variable, the variables in both datasets have to be called the same. If they are not, `append` creates an additional variable with all observations coming from the other file as missing.
- We only covered the "easy" types of `append`. Mitchell (2010) offers a discussion on which problems you may encounter and how to go around them.

3.7 Merging New Variables to Existing Dataset

The previous section on `append` concentrated on the situation when you need to add new observations to a dataset. This section concentrates on situations when you are looking to add new variables (often coming from another source) based on a match with a given variable (note that this section draws quite heavily on Mitchell (2010)). This procedure is done using the command `merge`.

Before we continue to specific examples, here is the terminology we have to understand:

- *Master data* - the dataset in memory, into which we want to merge new variables

- *Using data* - a dataset saved somewhere, from which we want to merge new variables.
- *Key variable(s)* - one or more variables that are matched in both datasets.

The syntax for merge is the following:

```
merge x:x varlist using filename [, options]
```

where

- `x:x` specifies the type of merge (we will cover 1:1, 1:m and m:1, see below)
- *varlist* is a list of variables on which the merge will be done (*key variable(s)*)
- *filename* is the *using* dataset (the *master* dataset has to be loaded in memory)

3.7.1 1:1 Merges

With a 1:1 merge, the merging variable has to uniquely identify observations in both the master and the using dataset. We use this in cases where we need to merge one instance of each dataset to each other. Let's now look at an example family data from Mitchell (2010):

```
. use dads1, clear

. list // notice that age, race and hs start with "d" prefix

     +---------------------------+
     | famid   dage   drace   dhs |
     |---------------------------|
  1. |     1     21       1     0 |
  2. |     2     25       1     1 |
  3. |     3     31       2     1 |
  4. |     4     25       2     1 |
     +---------------------------+

. use moms1, clear

. list // here they start with "m" prefix

     +---------------------------+
     | famid   mage   mrace   mhs |
     |---------------------------|
  1. |     1     33       2     1 |
  2. |     2     28       1     1 |
  3. |     3     24       2     1 |
  4. |     4     21       1     0 |
     +---------------------------+
```

We now have the *moms1* dataset in memory. Moreover, we can see that all families are uniquely identified using *famid* variable in both datasets. Therefore, we can merge the *dads1* dataset into *moms1* using the command

```
. merge 1:1 famid using dads1

    Result                           # of obs.
    -----------------------------------------
    not matched                             0
    matched                                 4  (_merge==3)
    -----------------------------------------

. list // Notice that everything was matched

     +-------------------------------------------------------------------+
     | famid   mage   mrace   mhs   dage   drace   dhs        _merge |
     |-------------------------------------------------------------------|
  1. |     1     33       2     1     21       1     0   matched (3) |
  2. |     2     28       1     1     25       1     1   matched (3) |
  3. |     3     24       2     1     31       2     1   matched (3) |
  4. |     4     21       1     0     25       2     1   matched (3) |
     +-------------------------------------------------------------------+
```

We can see that all the information is joined together for the four families. Notice the (very important) variable called `_merge` - this variable tells us the status of each observation.

In this case, all observations have `_merge` equal to 3 (it's a numeric value labeled variable). This is because all observations were found in both master and using dataset and were successfully matched.

Let's now look at a different example:

```
. use dads2, clear

. list // Now we have family 2, but 3 and 5 are missing

     +---------------------------+
     | famid   dage   drace   dhs |
     |---------------------------|
  1. |     1     21       1     0 |
  2. |     2     25       1     1 |
  3. |     4     25       2     1 |
     +---------------------------+

. use moms2, clear

. list // Here 2 is missing

     +---------------------------+
     | famid   mage   mrace   mhs |
     |---------------------------|
  1. |     1     33       2     1 |
  2. |     3     24       2     1 |
  3. |     4     21       1     0 |
  4. |     5     39       2     0 |
     +---------------------------+
```

Merging the two datasets,

```
. merge 1:1 famid using dads2

    Result                           # of obs.
    -----------------------------------------
    not matched                             3
        from master                         2  (_merge==1)
        from using                          1  (_merge==2)

    matched                                 2  (_merge==3)
    -----------------------------------------

. list // Now 3 families were not matched

     +-------------------------------------------------------------------+
     | famid   mage   mrace   mhs   dage   drace   dhs           _merge |
     |-------------------------------------------------------------------|
  1. |     1     33       2     1     21       1     0       matched (3) |
  2. |     3     24       2     1      .       .     .   master only (1) |
  3. |     4     21       1     0     25       2     1       matched (3) |
  4. |     5     39       2     0      .       .     .   master only (1) |
  5. |     2      .       .     .     25       1     1    using only (2) |
     +-------------------------------------------------------------------+
```

we can see that the values of variables which were not matched were set to missing and the information about the reason is included in `_merge`.

Sometimes, more than one variable is needed to merge the two files. Take the following datasets:

```
. use kids1, clear

. sort famid kidid

. list, sepby(famid) // There can be multiple kids per family

     +-----------------------------+
     | famid   kidid   kage   kfem |
     |-----------------------------|
  1. |     1       1      3      1 |
     |-----------------------------|
  2. |     2       1      8      0 |
  3. |     2       2      3      1 |
     |-----------------------------|
  4. |     3       1      4      1 |
  5. |     3       2      7      0 |
     |-----------------------------|
  6. |     4       1      1      0 |
  7. |     4       2      3      0 |
  8. |     4       3      7      0 |
     +-----------------------------+

. use kidname, clear
```

```
. sort famid kidid

. list, sepby(famid) // Here we have their names

     +----------------------+
     | famid   kidid   kname |
     |----------------------|
  1. |     1       1     Sue |
     |----------------------|
  2. |     2       1     Vic |
  3. |     2       2     Flo |
     |----------------------|
  4. |     3       1     Ivy |
  5. |     3       2     Abe |
     |----------------------|
  6. |     4       1     Tom |
  7. |     4       2     Bob |
  8. |     4       3     Cam |
     +----------------------+
```

If we want to attach names to these kids, we can do it in the following way:

```
.  use kids1, clear

. merge 1:1 famid kidid using kidname // merging on two variables

    Result                           # of obs.
    -----------------------------------------
    not matched                             0
    matched                                 8  (_merge==3)
    -----------------------------------------

. list // Everything was matched - we now have kid names in the dataset

     +-------------------------------------------------------+
     | famid   kidid   kage   kfem   kname        _merge     |
     |-------------------------------------------------------|
  1. |     1       1      3      1     Sue   matched (3)     |
  2. |     2       1      8      0     Vic   matched (3)     |
  3. |     2       2      3      1     Flo   matched (3)     |
  4. |     3       1      4      1     Ivy   matched (3)     |
  5. |     3       2      7      0     Abe   matched (3)     |
     |-------------------------------------------------------|
  6. |     4       1      1      0     Tom   matched (3)     |
  7. |     4       2      3      0     Bob   matched (3)     |
  8. |     4       3      7      0     Cam   matched (3)     |
     +-------------------------------------------------------+
```

3.7.2 1:m Merges

Suppose now we want to merge kids to moms. As moms can have multiple kids, the `merge 1:1` command would return an error because the key variable would not uniquely identify observations in the using dataset:

```
.  use moms1, clear

. merge 1:1 famid using kids1 // returns an error
variable famid does not uniquely identify observations in the using data
r(459);
```

Luckily, we can use the `1:m` merge where there is no such requirement:[4]

```
.  merge 1:m famid using kids1 // we can see that everything was matched

    Result                           # of obs.
    -----------------------------------------
    not matched                             0
    matched                                 8  (_merge==3)
    -----------------------------------------

. sort famid kidid

. list, sepby(famid)

     +-------------------------------------------------------------------+
     | famid   mage   mrace   mhs   kidid   kage   kfem        _merge |
     |-------------------------------------------------------------------|
  1. |     1     33       2     1       1      3      1   matched (3) |
     |-------------------------------------------------------------------|
  2. |     2     28       1     1       1      8      0   matched (3) |
  3. |     2     28       1     1       2      3      1   matched (3) |
     |-------------------------------------------------------------------|
  4. |     3     24       2     1       1      4      1   matched (3) |
  5. |     3     24       2     1       2      7      0   matched (3) |
     |-------------------------------------------------------------------|
  6. |     4     21       1     0       1      1      0   matched (3) |
  7. |     4     21       1     0       2      3      0   matched (3) |
  8. |     4     21       1     0       3      7      0   matched (3) |
     +-------------------------------------------------------------------+
```

Let's now quickly look at an example where there is no perfect match:

```
.  use moms2, clear // remember here famid 2 is missing

. merge 1:m famid using kids2 // similar as kids1, but famid 1 and 5 are missing

    Result                           # of obs.
```

[4]Note that as we perform a `1:m` merge, the requirement for the *master* dataset still holds.

```
    ----------------------------------------
    not matched                             4
        from master                         2  (_merge==1)
        from using                          2  (_merge==2)

    matched                                 5  (_merge==3)
    ----------------------------------------

. sort famid kidid

. list, sepby(famid) // some observations were not matched

     +-------------------------------------------------------------------------+
     | famid   mage   mrace   mhs   kidid   kage   kfem            _merge |
     |-------------------------------------------------------------------------|
  1. |     1     33       2     1       .      .      .   master only (1) |
     |-------------------------------------------------------------------------|
  2. |     2      .       .     .       1      8      0    using only (2) |
  3. |     2      .       .     .       2      3      1    using only (2) |
     |-------------------------------------------------------------------------|
  4. |     3     24       2     1       1      4      1       matched (3) |
  5. |     3     24       2     1       2      7      0       matched (3) |
     |-------------------------------------------------------------------------|
  6. |     4     21       1     0       1      1      0       matched (3) |
  7. |     4     21       1     0       2      3      0       matched (3) |
  8. |     4     21       1     0       3      7      0       matched (3) |
     |-------------------------------------------------------------------------|
  9. |     5     39       2     0       .      .      .   master only (1) |
     +-------------------------------------------------------------------------+
```

3.7.3 m:1 Merges, `keepusing` Option and Sequential Merge

The `m:1` type is really similar to `1:m`, except in this case you need the observations in the *using* dataset to be uniquely identified. As it is similar, we will show two additional options here rather than just repeat what was done above. Suppose you have a list of kids in the class with their names (kidname) and you are only interested in how old their parents are. You can find out using the following sequence of commands:

```
.  use kidname, clear

. merge m:1 famid using moms1, keepusing(mage) // will only add mom's age

    Result                           # of obs.
    -----------------------------------------
    not matched                             0
    matched                                 8  (_merge==3)
    -----------------------------------------

. assert _merge==3 // check that everything was matched properly

. drop _merge // drop the merge variable
```

```
. list // mage was added (and nothing else)

     +-----------------------------+
     | famid   kidid   kname   mage |
     |-----------------------------|
  1. |     1       1     Sue     33 |
  2. |     2       2     Flo     28 |
  3. |     2       1     Vic     28 |
  4. |     3       1     Ivy     24 |
  5. |     3       2     Abe     24 |
     |-----------------------------|
  6. |     4       1     Tom     21 |
  7. |     4       2     Bob     21 |
  8. |     4       3     Cam     21 |
     +-----------------------------+

. merge m:1 famid using dads1, keepusing(dage) // add dad's age only

    Result                           # of obs.
    -----------------------------------------
    not matched                             0
    matched                                 8  (_merge==3)
    -----------------------------------------

. assert _merge==3 // check that everything was matched

. drop _merge // drop the merging one

. list // now both mage and dage are there

     +-------------------------------------+
     | famid   kidid   kname   mage   dage |
     |-------------------------------------|
  1. |     1       1     Sue     33     21 |
  2. |     2       2     Flo     28     25 |
  3. |     2       1     Vic     28     25 |
  4. |     3       1     Ivy     24     31 |
  5. |     3       2     Abe     24     31 |
     |-------------------------------------|
  6. |     4       1     Tom     21     25 |
  7. |     4       2     Bob     21     25 |
  8. |     4       3     Cam     21     25 |
     +-------------------------------------+
```

Notes to `merge`:

- In the last example, we were lucky that everything was matched so we could drop the `_merge` variable. Generally, we could rename this variable to a different name and continue with the sequential merge, specify a different name for the information variable (option `generate(`*`varname`*`)`), or tell Stata to not generate (not recommended). If you do not use any of these three possibilities, Stata will return an error.

- There is one more type of merge, `m:m`. However, it is not recommended to use this one, as it is not stable and very often does not produce the correct output. Its use is discouraged by Stata itself. If you need to perform many-to-many merge, use the command `joinby` instead. Alternatively, there is very likely an option to use `m:1` or `1:m` merge that you do not yet know about - see the "Troubleshooting m:m merges" section of the pdf help file for `merge` for more information.
- We have not covered usual problems that may arise with `merge`. Mitchell (2010) covers these in detail.

References

Kohler, U. & Kreuter, F. (2012). *Data Analysis Using Stata, Third Edition*. Taylor & Francis.

Mitchell, M. N. (2010). *Data Management Using Stata: A Practical Handbook*. StataCorp LP.