# *Programming, Data Management and Visualization*

## Module B: **Programming preliminaries**

**Alexander Ahammer**

Department of Economics, Johannes Kepler University, Linz, Austria
Christian Doppler Laboratory Ageing, Health, and the Labor Market, Linz, Austria

$\gamma$ version, final
Last updated: Monday 2nd December, 2019 (11:17)


# Introduction

- In this module we talk more specifically about programming. We cover functions, macros, lists, scalars, and matrices.

- You should already be familiar with functions. Macros and scalars are often overlooked, although familiarity is useful to write do-files more efficiently.

- In the end you will (hopefully)
  - understand varlists, numlists, and `if` and `in` qualifiers
  - know how to handle missing data
  - be familiar with functions for use with `generate`
  - be familiar with the capabilities of `egen`
  - know how to use by-groups effectively
  - understand the use of local and global macros
  - understand how to use scalars
  - know how to use matrices

# B.1

*Lists and logical qualifiers*

# varlists

- Many Stata commands accept a varlist, a list of one or more variables that can contain **variable names or wild cards,** e.g.,
  - `*` is an arbitrary set of characters
  - `?` is a single arbitrary character
  - `-` can be used to specify a hyphenated list

- Make sure to be aware of the order of variables (e.g., for a hyphenated list), it is shown in the variables window and can be checked with `describe`.

```
. des p_* sl_* e_start-e_tenure
              storage   display    value
variable name   type    format     label      variable label
------------------------------------------------------------------------------
p_age           float   %9.0g                 [worker] age in years
p_female        byte    %8.0g                 [worker] =1 if female
p_educ          byte    %27.0g     educ       [worker] education
sl_start        int     %td                   [sick leave] start date
sl_end          int     %td                   [sick leave] end date
sl_dur          byte    %9.0g                 [sick leave] duration
e_start         int     %d                    [emp] start date
e_end           int     %d                    [emp] end date
e_class         byte    %19.0g     classlab   [emp] occupation
e_tenure        int     %9.0g                 [emp] job tenure
```

# numlists

- A numlist is a **list of numeric arguments** which can be provided in several ways.
  - It can be spelled out explicitly: `0.5 1 1.5 2 2.5 3`
  - It can involve a range of values, such as 1/4 or −3/3 (these lists would include the integers between those limits)
  - You can also specify `10 15 to 30` or, equivalently, `10 15:30` which would count from 10 to 30 by 5s.
  - You can count by steps, for example, 1 (2) 9, which is a list of the first five odd integers, or 9 (−2) 1, which is the same list in reverse order. Square brackets can be used instead of parentheses.
- Avoid commas inside numlists, which will cause errors. In other languages an interval from 1 to 10 may be spelled 1,10 — in Stata it's 1/10.

# `if` *exp* and `in` *range* qualifiers

- By default, Stata commands operate on all observations in memory. Almost all commands, however, accept qualifiers that restrict the command to a subset of observations.
  - `if` *exp* for logical conditions
  - `in` *range* for observation numbers

```
. su e_wage in 1/10       /* average wage ONLY for the first 10 observations */
    Variable |        Obs        Mean    Std. Dev.       Min        Max
-------------+---------------------------------------------------------
      e_wage |         10    13701.51    6465.033       8471      28529

. su e_wage if p_female == 1
    Variable |        Obs        Mean    Std. Dev.       Min        Max
-------------+---------------------------------------------------------
      e_wage |    130,096    19632.96    11493.88   .0033333   323810.7
```

# Missing data

- Stata possesses 27 numeric missing value codes, the system missing value `.` and 26 others from `.a` to `.z`.
  - Possible to indicate different reasons for missing values.
  - E.g., `.r` can me used to indicate that the person refused to answer.

- **IMPORTANT** Missings are treated as **very large positive numbers!**
  - `age > 60` is true whenever age has a value greater than 60 *or is missing*.
  - Important when generating dummy variables!
  - `g age_plus60 = age > 60 & !missing(age)`

- Use the `missing()` function, which returns 1 if any of its arguments is missing and 0 otherwise. You can negate it with `!`, i.e., `!missing()`.

- Always check whether a variable has missings before manipulating it or performing computations!

# Missing data

- Typically Stata omits observations with missing data from any computation ($\longrightarrow$ casewise deletion). For `generate` and `replace`, missing values are propagated so that any function of missing data is missing as well.
  - Some exceptions exist. For example, functions `max()`, `min()`, and the `egen` rowwise functions **ignore** missing values!
  - For example, `rowmean(x1,x2,x3)` will only return missing when all three vars are missing, otherwise it will compute the mean using only the non-missing vars.
- For strings, the empty or null string `""` is taken as missing. The `missing()` function can be used as well.
- The commands `mvdecode` and `mvencode` can be helpful when you want to recode certain numeric values (e.g., 99) as missing.
- More information in the manual
  - `help missing values` [12.2.1 Missing values]
  - `help mvencode`

# Missing data

```
. replace p_educ = .m if missing(p_educ)
(33,286 real changes made, 33,286 to missing)
. list p_educ in 1/5
```

| | p_educ |
|---|---|
| 1. | Lehre |
| 2. | Lehre |
| 3. | Lehre |
| 4. | .m |
| 5. | Pflichtschule |

```
. ta p_educ, nola
```

| [worker] education | Freq. | Percent | Cum. |
|---|---|---|---|
| 0 | 1,231 | 0.43 | 0.43 |
| 1 | 37,108 | 12.84 | 13.26 |
| 2 | 122,264 | 42.29 | 55.55 |
| 3 | 50,449 | 17.45 | 73.01 |
| 4 | 59,636 | 20.63 | 93.63 |
| 5 | 18,401 | 6.37 | 100.00 |
| Total | 289,089 | 100.00 | |

```
. * 289,089 non-missing obs.
. count if p_educ >= 0    /* missings are large numbers! */
  322,375
. count if !missing(p_educ)
  289,089
```

# B.2

## *Functions*

# Helpful functions for the `generate` command

- Fundamental commands for data transformation: `generate` (create new vars) and `replace` (change contents of existing vars).
- `inlist()` and `inrange()` can be particularly helpful.
  - `inlist()` returns 1 if the variable matches one of the elements of the list.
  - `inrange()` returns 1 if the values of a variable fall within a real interval
- `round()`, `int()`, or `floor()` can be useful when integers instead of decimal values are needed (for example to generate categorial vars).
- `sum()` produces cumulative or running sums (as opposed to `egen`'s `total()` function).

# Helpful functions for the `generate` command

```
. g f_ind_farm = inlist(f_industry,1,4,29)        /* also works with strings! */
. g p_age_mid = inrange(p_age,25,45)
. g p_age_cat = round(p_age)
. g sl_dur_sum = sum(sl_dur)
. list f_ind_farm p_age_mid p_age p_age_cat sl_dur sl_dur_sum in 1/10, sep(0)
```

| | f_ind_~m | p_age_~d | p_age | p_age_~t | sl_dur | sl_dur~m |
|---|---|---|---|---|---|---|
| 1. | 0 | 0 | 58.5 | 59 | 8 | 8 |
| 2. | 0 | 0 | 58.91667 | 59 | 5 | 13 |
| 3. | 0 | 0 | 59.83333 | 60 | 4 | 17 |
| 4. | 0 | 0 | 55.75 | 56 | 9 | 26 |
| 5. | 0 | 0 | 49.5 | 50 | 3 | 29 |
| 6. | 0 | 0 | 49.91667 | 50 | 1 | 30 |
| 7. | 0 | 0 | 50.33333 | 50 | 7 | 37 |
| 8. | 0 | 0 | 51 | 51 | 12 | 49 |
| 9. | 0 | 0 | 51 | 51 | 7 | 56 |
| 10. | 0 | 0 | 52.66667 | 53 | 4 | 60 |

# *Recap* Using `if` *exp* with indicator vars

- A key element of data preparation is generating **indicator** (dummy) **variables,** a var that takes on $(0, 1)$ depending on whether a condition is satisfied.
- Dummy vars are generated according to a *Boolean* condition; an expression that evaluates to true (1) or false (0) for every obs in the data.
- Always take care of **missing values!** The safe way is to always use the `missing()` function irrespective of whether the var actually has missings.
- Suppose we want to generate a var indicating whether a person is a blue collar worker in `pdmv_sl.dta`. There are two ways of doing that:

**(1) The tedious way**

```
g e_bc = .
replace e_bc = 1 if e_class == 1 & !missing(e_class)
replace e_bc = 0 if e_class != 1 & !missing(e_class)
```

**(2) The fast and *efficient* way**

```
g e_bc = e_class == 1 if !missing(e_class)
```

# The `cond()` function

- You may want to code a variable as $a$ if a condition is true and $b$ if that condition is false. In this case you use the `cond(`$x$`,` $a$`,` $b$`)` function.
  - $x$ is the condition to be tested
  - $a$ is the result when true
  - $b$ is the result when false

- Suppose you want to generate a categorical variable which is 1 when the *daily* wage of a worker is above EUR 2,500 and 2 else.

```
g e_highwage = cond(e_wage/12 > 2500,1,2)
la def wageval 1 "m wage > 2,500" 2 "m wage <= 2,500"
la val e_highwage wageval
```

  $\Longrightarrow$ Saves time because you don't have to generate the parameter you want to condition on (i.e., the monthly wage) separately.

- Conditions can also be nested. That is, the second and third argument can be additional `cond()` functions.

# The `cond()` function

```
. g e_highwage = cond(e_wage/12 > 2500,1,2)
. la def wageval 1 "m wage > 2,500" 2 "m wage <= 2,500"
. la val e_highwage wageval
. tabstat sl_dur, by(e_highwage) s(mean sd)
Summary for variables: sl_dur
     by categories of: e_highwage
     e_highwage |      mean        sd
----------------+--------------------
 m wage > 2,500 |  5.990518   5.14987
m wage <= 2,500 |  6.036301  5.503043
----------------+--------------------
          Total |  6.020889  5.386769
----------------------------------------
```

# Recoding discrete and continuous vars

- `recode` creates a new variable based on the coding of an existing **categorical** variable. Use it to combine data transformations that involve many similar statements.
- Do not use the line-by-line approach as in method (1) below, this is prone to errors (especially when constructing a large number of statements).
- Check `help recode` for its syntax. You use equality signs to indicate assignment: *oldvalue(s)* $\longrightarrow$ *newvalue*. The former allows for varlists.

**(1) The inefficient method**

```
g f_sector = .
replace f_sector = 1 if f_industry == 1
replace f_sector = 2 if f_industry == 4
replace f_sector = 3 if inrange(f_industry,5,18)
...
replace f_sector = . if f_industry == 9
```

**(2) The efficient method**

```
recode f_industry (1 = 1) (4 = 2) (5/18 = 3) ... (99 = .), gen(f_sector)
```

# Recoding discrete and continuous vars

```
. fre f_industry, rows(5)
f_industry ── [firm] NACE95 industry
```

| | | Freq. | Percent | Valid | Cum. |
|---|---|---|---|---|---|
| Valid | 1 AA | 2381 | 0.74 | 0.74 | 0.74 |
| | 4 CB | 1012 | 0.31 | 0.31 | 1.05 |
| | : | : | : | : | : |
| | 29 OA | 12010 | 3.73 | 3.73 | 97.46 |
| | 99 | 8182 | 2.54 | 2.54 | 100.00 |
| | Total | 322375 | 100.00 | 100.00 | |

```
. recode f_industry (1 = 1) (4 = 2) (5/18 = 3) (19 = 4) (20 = 5) (21 = 6) ///
>         (22 = 7) (23 = 8) (24 = 9) (25 = 10) (26 = 11) (27 = 12) (28 = 13) ///
>         (29 = 14) (99 = .), gen(f_sector)
(319994 differences between f_industry and f_sector)
. la def sectorval 1 "A" 2 "C" 3 "D" 4 "E" 5 "F" 6 "G" 7 "H" 8 "I" 9 "J" ///
>                 10 "K" 11 "L" 12 "M" 13 "N" 14 "O"
. la val f_sector sectorval
. fre f_sector, rows(8)
f_sector ── RECODE of f_industry ([firm] NACE95 industry)
```

| | | Freq. | Percent | Valid | Cum. |
|---|---|---|---|---|---|
| Valid | 1 A | 2381 | 0.74 | 0.76 | 0.76 |
| | 2 C | 1012 | 0.31 | 0.32 | 1.08 |
| | 3 D | 101082 | 31.36 | 32.17 | 33.25 |
| | 4 E | 3144 | 0.98 | 1.00 | 34.25 |
| | : | : | : | : | : |
| | 11 L | 23255 | 7.21 | 7.40 | 88.45 |
| | 12 M | 5573 | 1.73 | 1.77 | 90.23 |
| | 13 N | 18702 | 5.80 | 5.95 | 96.18 |
| | 14 O | 12010 | 3.73 | 3.82 | 100.00 |
| | Total | 314193 | 97.46 | 100.00 | |
| Missing | . | 8182 | 2.54 | | |
| Total | | 322375 | 100.00 | | |

# Recoding discrete and continuous vars

- For continuous vars it depends on whether you want to use specific **threshold values** for categorization.

- Generally, you want to **avoid defining arbitrary thresholds.** However, sometimes there are institutional circumstances that give rise to thresholds.
  - For example, the Austrian unemployment office determines that a *mass layoff* (ML) is given when 5 out of 20–100 employees in a firm are laid off.
  - For such cases, it makes sense to use the generate functions `recode()` or `irecode()` to cut a continuous var (e.g., ML size) at specific values.

- Without an institutional prior, either cut the variable at **constant intervals** (e.g., age in 5 equally sized intervals) or into **percentiles.**
  - `egen`'s `cut` function and the `at` option allows a numlist to define an interval
  - you can specify the increments manually in `irecode()`
  - to cut in constant *number of intervals,* use the `generate` function `autocode`
  - to cut in percentiles, use the `xtile` command

# Recoding discrete and continuous vars

```
. su p_age, det

                    [worker] age in years
-------------------------------------------------------------
      Percentiles      Smallest
 1%     18.91667            18
 5%         20.5            18
10%           22            18       Obs               322,375
25%     26.83333            18       Sum of Wgt.       322,375
50%         36.5                     Mean            36.82896
                       Largest       Std. Dev.       11.19948
75%     45.91667            65
90%     52.41667            65       Variance        125.4284
95%     55.33333            65       Skewness        .1609392
99%     59.08333            65       Kurtosis        1.908309

.
. // cut according to arbitrary thresholds
. g p_agecat = irecode(p_age,25,40,60)
. la def agecatval1 0 "a<=25" 1 "25<a<=40" 3 "40<a<=60" 4 "a>60"
. la val p_agecat agecatval1
. tabstat p_age, s(mean min max) by(p_agecat)
Summary for variables: p_age
     by categories of: p_agecat
p_agecat |      mean       min       max
---------+--------------------------------
   a<=25 |  21.92499        18        25
25<a<=40 |  32.36783  25.08333        40
       2 |  48.17418  40.08333        60
40<a<=60 |  61.25673  60.08333        65
---------+--------------------------------
   Total |  36.82896        18        65
-------------------------------------------
```

# Recoding discrete and continuous vars

```
. // cut in 5-year intervals
. su p_age
```

| Variable | Obs | Mean | Std. Dev. | Min | Max |
|---|---|---|---|---|---|
| p_age | 322,375 | 36.82896 | 11.19948 | 18 | 65 |

```
. egen p_agecat3 = cut(p_age), at(18(5)68)
. * if you use 65 as the maximum, you will get 139 missing values
. tabstat p_age, s(mean min max) by(p_agecat3)
Summary for variables: p_age
     by categories of: p_agecat3
```

| p_agecat3 | mean | min | max |
|---|---|---|---|
| 18 | 20.89855 | 18 | 22.91667 |
| 23 | 25.3849 | 23 | 27.91667 |
| 28 | 30.39294 | 28 | 32.91667 |
| 33 | 35.48379 | 33 | 37.91667 |
| 38 | 40.47929 | 38 | 42.91667 |
| 43 | 45.4373 | 43 | 47.91667 |
| 48 | 50.33335 | 48 | 52.91667 |
| 53 | 55.19573 | 53 | 57.91667 |
| 58 | 59.35504 | 58 | 62.91667 |
| 63 | 63.88849 | 63 | 65 |
| Total | 36.82896 | 18 | 65 |

# Recoding discrete and continuous vars

```
. // cut age in 5 equally sized intervals from 18 to 65
. g p_agecat4 = autocode(p_age,5,18,65)
. tabstat p_age, s(mean min max) by(p_agecat4)
Summary for variables: p_age
     by categories of: p_agecat4
p_agecat4 |      mean       min       max
----------+------------------------------
27.399999 |  23.01836        18  27.33333
36.799999 |  31.97151  27.41667     36.75
46.200000 |  41.52567  36.83333  46.16667
55.599998 |  50.45601     46.25  55.58333
       65 |  57.82601  55.66667        65
----------+------------------------------
    Total |  36.82896        18        65
----------------------------------------
.
. // cut age in 5 quintiles
. xtile p_agecat5 = p_age, n(5)
. tabstat p_age, s(mean min max) by(p_agecat5)
Summary for variables: p_age
     by categories of: p_agecat5 (5 quantiles of p_age)
p_agecat5 |      mean       min       max
----------+------------------------------
        1 |  22.00738        18  25.16667
        2 |  28.75398     25.25      32.5
        3 |  36.51426  32.58333  40.33333
        4 |  44.13008  40.41667  47.91667
        5 |  52.94069        48        65
----------+------------------------------
    Total |  36.82896        18        65
----------------------------------------
```

# Functions for the `egen` command

- `egen` is a powerful command for data preparation that allows for many functions which cannot be used with `generate` and `replace`.
- Most importantly, **rowwise calculations** can only be done with `egen`. That is, calculating sums, averages, standard deviations, extrema, and counts **across variables** for every obs.
  - Row functions include `rowmean()`, `rowmax()`, `rowmin()`, `rowtotal()`, etc.
  - Wildcards are allowed, `rowmean(pop*)` instead of `rowmean(pop80,pop90)`.
  - Don't forget $\longrightarrow$ missing values are ignored!
- Another set of functions calculate certain statistics for by-groups in the data (discussed in the next subsection).
  - `count()`, `min()`, `max()`, `total()` are typically useful.
  - Most allow for `by:` prefix as well as the `, by()` option (equivalent!)
- Many more functions are available (esp. all basic statistics such as median, kurtosis, skewness, percentiles, etc.), but also helpful functions for data preparation (e.g., `group` or `tag`). Check `help egen` for a detailed overview.
- It's also useful to download the user-written `egenmore` command from SSC, which extends `egen` with further functions.

# Functions for the `egen` command

```
. * average age of workers in the panel
. by id_worker: egen avgage = mean(p_age)
. list id_worker sl_start p_age avgage if inrange(id_worker,1738,1788), sepby(id_worker)
```

| | id_wor~r | sl_start | p_age | avgage |
|---|---|---|---|---|
| 30. | 1738 | 18feb2005 | 38.91667 | 41.90625 |
| 31. | 1738 | 08mar2006 | 40 | 41.90625 |
| 32. | 1738 | 27dec2006 | 40.75 | 41.90625 |
| 33. | 1738 | 20feb2007 | 40.91667 | 41.90625 |
| 34. | 1738 | 27feb2007 | 40.91667 | 41.90625 |
| 35. | 1738 | 31dec2008 | 42.75 | 41.90625 |
| 36. | 1738 | 21oct2010 | 44.58333 | 41.90625 |
| 37. | 1738 | 10aug2012 | 46.41667 | 41.90625 |
| 38. | 1788 | 25jan2005 | 53.25 | 54.02083 |
| 39. | 1788 | 06may2005 | 53.58333 | 54.02083 |
| 40. | 1788 | 21feb2006 | 54.33333 | 54.02083 |
| 41. | 1788 | 13sep2006 | 54.91667 | 54.02083 |

```
. * doesn´t make sense, but just in contrast: row min of p_age and avgage
. egen TESTmin = rowmin(p_age avgage)
. list id_worker sl_start p_age avgage TESTmin if id_worker == 1788
```

| | id_wor~r | sl_start | p_age | avgage | TESTmin |
|---|---|---|---|---|---|
| 38. | 1788 | 25jan2005 | 53.25 | 54.02083 | 53.25 |
| 39. | 1788 | 06may2005 | 53.58333 | 54.02083 | 53.58333 |
| 40. | 1788 | 21feb2006 | 54.33333 | 54.02083 | 54.02083 |
| 41. | 1788 | 13sep2006 | 54.91667 | 54.02083 | 54.02083 |

# B.3

*By-groups and observation numbering*

# Computations for by-groups

- One of Stata's most useful features is to **compute statistics or transform variables over by-groups.** These are identified via the `by` *varlist* `:` prefix.

- Using the prefix with one or more **categorical** variables, a command will be repeated automatically for each value of the `by` *varlist*.
  - If *varlist* contains more than one categorical var, by will be executed on every possible unique combination of the vars.

- By specifying `bysort` *varlist*, you can automatically sort the data by *varlist* in ascending order.
  - This is *necessary* if you did not sort the data according to *varlist* first, since
  - the command can only be executed if the data are sorted according to by-groups.

- It also allows a prefix such as `bys` *varlist1* `(`*varlist2*`)`: which first sorts the data according to *varlist1* and *varlist2*, and then executes the command for every distinct realization of the variables in *varlist1*.

  $\Longrightarrow$ *varlist2* is only used for sorting, not to build the by-group!

# Computations for by-groups

## Executing summarize command for different by-groups

```
. sort e_class p_female   /* data have to be sorted first, otherwise use bys instead of by */
. by e_class p_female: su sl_dur
```

```
-> e_class = blue collar worker, p_female = 0
    Variable |        Obs        Mean    Std. Dev.       Min        Max
-------------+---------------------------------------------------------
      sl_dur |    141,464    6.345671     5.54329          1         44

-> e_class = blue collar worker, p_female = 1
    Variable |        Obs        Mean    Std. Dev.       Min        Max
-------------+---------------------------------------------------------
      sl_dur |     54,296    6.546836    5.951415          1         44

-> e_class = white collar worker, p_female = 0
    Variable |        Obs        Mean    Std. Dev.       Min        Max
-------------+---------------------------------------------------------
      sl_dur |     50,815    5.402263    4.753275          1         44

-> e_class = white collar worker, p_female = 1
    Variable |        Obs        Mean    Std. Dev.       Min        Max
-------------+---------------------------------------------------------
      sl_dur |     75,800    5.452731    4.956985          1         44
```

# Computations for by-groups

## Data transformations for different by-groups

```
. g year = yofd(sl_start)
. bys id_worker (year sl_start): egen sl_totaldur = total(sl_dur)
. list id_worker year sl_start sl_dur sl_totaldur if id_worker == 8246, sepby(year)
```

| | id_wor~r | year | sl_start | sl_dur | sl_tot~r |
|---|---|---|---|---|---|
| 223. | 8246 | 2006 | 15feb2006 | 5 | 34 |
| 224. | 8246 | 2007 | 19sep2007 | 3 | 34 |
| 225. | 8246 | 2008 | 28apr2008 | 3 | 34 |
| 226. | 8246 | 2010 | 10feb2010 | 3 | 34 |
| 227. | 8246 | 2010 | 11oct2010 | 5 | 34 |
| 228. | 8246 | 2010 | 18oct2010 | 3 | 34 |
| 229. | 8246 | 2011 | 02may2011 | 5 | 34 |
| 230. | 8246 | 2011 | 24oct2011 | 5 | 34 |
| 231. | 8246 | 2011 | 10nov2011 | 2 | 34 |

# Computations for by-groups

## Data transformations for different by-groups

```
. bys id_worker year (sl_start): egen sl_totaldur2 = total(sl_dur)
. list id_worker year sl_start sl_dur sl_totaldur sl_totaldur2 if id_worker == 8246, sepby(year)
```

| | id_wor~r | year | sl_start | sl_dur | sl_tot~r | sl_tot~2 |
|---|---|---|---|---|---|---|
| 223. | 8246 | 2006 | 15feb2006 | 5 | 34 | 5 |
| 224. | 8246 | 2007 | 19sep2007 | 3 | 34 | 3 |
| 225. | 8246 | 2008 | 28apr2008 | 3 | 34 | 3 |
| 226. | 8246 | 2010 | 10feb2010 | 3 | 34 | 11 |
| 227. | 8246 | 2010 | 11oct2010 | 5 | 34 | 11 |
| 228. | 8246 | 2010 | 18oct2010 | 3 | 34 | 11 |
| 229. | 8246 | 2011 | 02may2011 | 5 | 34 | 12 |
| 230. | 8246 | 2011 | 24oct2011 | 5 | 34 | 12 |
| 231. | 8246 | 2011 | 10nov2011 | 2 | 34 | 12 |

# Observation numbering

- When you refer to an observation, you can do this with its **observation number.** Observation numbers can be altered with `sort`.
  - `_N` is the highest observation number (total number of obs)
  - `_n` is the current observation number

- Under a by-group, `_N` is the total number of observations or the last observation *in the group* and `_n` is the current observation *of the group.*

- Understanding observation numbering is crucial, possible applications are manifold. For example,
  - Counting entries in a group (e.g., how many sick leaves per person?)
  - Identifying first and last observations in a group
  - Identifying obs with min or max values in a group
  - Identifying first and last spells in duration data

- **TIPP** If you use observation numbering always make sure to sort your data before. In particular, use the parentheses in the `bys` *`var (var2)`*`:` prefix.

# Observation numbering

```
. bys id_worker (sl_start): g n = _n
. bys id_worker (sl_start): g N = _N
. g n_OVR = _n
. g N_OVR = _N
. list id_worker sl_start n N *_OVR in 1/10, sepby(id_worker)
```

| | id_wor~r | sl_start | n | N | n_OVR | N_OVR |
|---|---|---|---|---|---|---|
| 1. | 166 | 04jun2011 | 1 | 3 | 1 | 322375 |
| 2. | 166 | 09nov2011 | 2 | 3 | 2 | 322375 |
| 3. | 166 | 20oct2012 | 3 | 3 | 3 | 322375 |
| 4. | 276 | 15jan2009 | 1 | 1 | 4 | 322375 |
| 5. | 548 | 18feb2005 | 1 | 6 | 5 | 322375 |
| 6. | 548 | 04jul2005 | 2 | 6 | 6 | 322375 |
| 7. | 548 | 05dec2005 | 3 | 6 | 7 | 322375 |
| 8. | 548 | 09aug2006 | 4 | 6 | 8 | 322375 |
| 9. | 548 | 28aug2006 | 5 | 6 | 9 | 322375 |
| 10. | 548 | 10apr2008 | 6 | 6 | 10 | 322375 |

```
.
. * number of sick leaves per worker and year
. bys id_worker year: g sl_N = _N
. su sl_N
```

| Variable | Obs | Mean | Std. Dev. | Min | Max |
|---|---|---|---|---|---|
| sl_N | 322,375 | 2.629035 | 1.933495 | 1 | 45 |

# Observation numbering

```
. // you can use observation numbering also to refer to specific cells in the data
. di id_worker[950]       /* id_worker of the 950th observation in the data */
29498
. di id_worker[_N]        /* id_worker of the last observation in the data */
14149633
. bys id_worker: g sl_dur_pre     = sl_dur[_n-1]
(52,739 missing values generated)
. bys id_worker: g sl_dur_first   = sl_dur[1]
. bys id_worker: g sl_dur_last    = sl_dur[_N]
. list id_worker sl_dur* if id_worker <= 548, sepby(id_worker)
```

| | id_wor~r | sl_dur | sl_dur~e | sl_d~rst | sl_d~ast |
|---|---|---|---|---|---|
| 1. | 166 | 5 | . | 5 | 4 |
| 2. | 166 | 8 | 5 | 5 | 4 |
| 3. | 166 | 4 | 8 | 5 | 4 |
| 4. | 276 | 9 | . | 9 | 9 |
| 5. | 548 | 12 | . | 12 | 7 |
| 6. | 548 | 7 | 12 | 12 | 7 |
| 7. | 548 | 1 | 7 | 12 | 7 |
| 8. | 548 | 4 | 1 | 12 | 7 |
| 9. | 548 | 3 | 4 | 12 | 7 |
| 10. | 548 | 7 | 3 | 12 | 7 |

# Observation numbering

## Exercise

**Exercise: Employment spells**

Use `pdmv_sl.dta` and generate a running indicator for employment spells in the data. Make sure that the data are sorted by `id_worker sl_start`, and use observation numbering and by-groups to indicate distinct spells for every single `id_firm` firm episode for every worker. For example,

| id_worker | id_firm | spell |
|---|---|---|
| 1 | A | 1 |
| 1 | A | 1 |
| 1 | B | 2 |
| 1 | B | 2 |
| 1 | C | 3 |
| 2 | D | 1 |

Compute also the number of sick leaves in the data per employment spell.

# Observation numbering

## Exercise

```
. use "data/pdmv_sl.dta", clear
(All sick leaves 2004-2012 for 10% sample of Austrian employees)
. sort id_worker sl_start
. by id_worker: g start = id_firm != id_firm[_n-1]
. by id_worker: g spell = sum(start)
. bys id_worker spell (sl_start): g sl_n = _N
. list id_firm sl_start start spell sl_n if id_worker == 1738, sepby(spell)
```

| | id_firm | sl_start | start | spell | sl_n |
|---|---|---|---|---|---|
| 30. | 1416048031600 | 18feb2005 | 1 | 1 | 3 |
| 31. | 1416048031600 | 08mar2006 | 0 | 1 | 3 |
| 32. | 1416048031600 | 27dec2006 | 0 | 1 | 3 |
| 33. | 1410075312000 | 20feb2007 | 1 | 2 | 5 |
| 34. | 1410075312000 | 27feb2007 | 0 | 2 | 5 |
| 35. | 1410075312000 | 31dec2008 | 0 | 2 | 5 |
| 36. | 1410075312000 | 21oct2010 | 0 | 2 | 5 |
| 37. | 1410075312000 | 10aug2012 | 0 | 2 | 5 |

# B.4

## *Macros*

# Macros

- Perhaps one of the most important concepts for programming are **macros.** A macro is a **variable container** that can hold either one object (e.g., a number, varname, or a string) or a *set of objects.*

- The Stata macro is an **alias** that contains a **name** and a **value.**
  - When its name is addressed, it returns the specified value.
  - That operation can be carried out at any time.
  - The macro's value can be modified with an additional command.

- Depending on its scope, the macro can be **local** or **global.** A local macro is ceases to exist when a do-file terminates, a global macro exists for the duration of the Stata program or an interactive session.

- A local macro can be defined using `local` *`macroname value`*, a global macro using `global` *`macroname value`*. The local macro is addressed with ‘*`macroname`*’, the global with $*`macroname`*.
  - If *`value`* is numeric, use an = in front
  - If *`value`* is a string, enclose it in `" "`

# Macros

## How macros are defined and addressed

```
.         loc hello "world"
.         di "`hello´"
world
.         loc x = 2
.         di `x´
2
.
.         global y = 3
.         di $y
3
```

# Macros

## How macros are defined and addressed

- Macros are addressed by the *macroname* inside left single-quote character (‘) and the right single-quote character (’). Different quotes are used to signify beginning and end of a macro when macros are nested.
  - `‘pid‘year’’` first addresses `‘year’`, then `‘pidyear’`.
- Correct punctuation is crucial when addressing macros. Apart from the apostrophes `‘ ’` that enclose the macro name, sometimes it’s required to enclose the entire expression in `" "`, for example if the macro contains a string and has to be logically evaluated against another string.
  - This could be the first line of an if-loop (see later): `if "‘var’" == "p_age"`
- Using an $=$ when defining the macro tells Stata to evaluate the remainder of the expression rather than merely aliased to the macro’s name.
  - `loc x = 1/2` will return 0.5 instead of `1/2` ($\longrightarrow$ evaluated!)
  - Do not use $=$ with strings or varnames, this can cause major problems.

# Macros

## Two examples

**Without equality sign**

```
. loc count 0
. loc anxlevel none mild moderate severe
. foreach a of local anxlevel {
  2.         loc count `count´ + 1
  3.         di "Anxiety level `count´: `a´"
  4. }
Anxiety level 0 + 1: none
Anxiety level 0 + 1 + 1: mild
Anxiety level 0 + 1 + 1 + 1: moderate
Anxiety level 0 + 1 + 1 + 1 + 1: severe
```

**With equality sign**

```
. loc count = 0
. loc anxlevel none mild moderate severe
. foreach a of local anxlevel {
  2.         loc count = `count´ + 1
  3.         di "Anxiety level `count´: `a´"
  4. }
Anxiety level 1: none
Anxiety level 2: mild
Anxiety level 3: moderate
Anxiety level 4: severe
```

- Note how we the first line inside the `foreach` loop $\longrightarrow$ it contains the local's name twice: once without punctation (which defines its name) and again after the $=$ sign, which addresses its current value.
- You can use also `loc ++count` instead of using `loc count = ‘count’ + 1` to increment the counter by 1.

# Macros

## Macro evaluation

- Macro evaluation generates macros on the fly. For example, you may need the outcome (or evaluated value) of a function, but since you **only need it once** it doesn't make sense to define a local macro.
- `‘= `*exp*`’` tells Stata to evaluate *exp* immediately.
- Suppose you have a time series graph and you want to draw a line on the x-axis at a particular date.
  - `xline(‘= mdy(1,1,2008)’)`
  - Stata will return `xline(17532)`
  - Easier than defining `loc xlineval = mdy(1,1,2008)`

# Macro functions

- Stata contains a lot of useful functions that allow you to retrieve and manipulate the contents of macros. A full list can be found in the manual, here I give an overview on important functions I use most often.
  - `help extended_fcn` [Manual link: Macros]

| Command | Example | Returns |
|---|---|---|
| `value label` *varname* | `loc lab: value label sl_dur` | String with value label of `sl_dur` |
| `variable label` *varname* | `loc lab: var label sl_dur` | String with variable label of `sl_dur` |
| `dir` *dir* `files` *path* | `loc folder: dir . files "*"` | String with all files stored in CWD |
| `display` | `loc x: display %9.4f sqrt(2)` | Results from display command |
| `word count` *string* | `loc wds: word count ‘v’` | Number of words in macro v |
| `word # of` *string* | `loc wd: word 1 of ‘v’` | First word in macro v |
| `length local` *macro* | `loc ln: length local ‘v’` | Length of macro v |

- Also sometimes useful: `levelsof` *var*`, local(`*mname*`)` stores all realizations of the variable *var* in the new local *mname*.

# Macro functions

## Exercise I

Exercise: **Using macro functions, part 1**

Use the data is `pdmv_sl.dta`. Change the `[sick leave]` prefix in the variable labels of all `sl_` variables to `[sl]`.

# Macro functions

## Exercise I

```
. des sl_*
              storage   display    value
variable name   type    format     label      variable label
------------------------------------------------------------------------------
sl_start        int     %td                   [sick leave] start date
sl_end          int     %td                   [sick leave] end date
sl_dur          byte    %9.0g                 [sick leave] duration
. foreach v of varlist sl_* {
  2.         loc oldlab : var label `v´
  3.                 di "`oldlab´"
  4.         loc labstr = substr("`oldlab´",strpos("`oldlab´","]")+2,.)
  5.                 di "`labstr´"
  6.         loc newlab = "[sl] " + "`labstr´"
  7.         la var `v´ "`newlab´"
  8. }
[sick leave] start date
start date
[sick leave] end date
end date
[sick leave] duration
duration
. des sl_*
              storage   display    value
variable name   type    format     label      variable label
------------------------------------------------------------------------------
sl_start        int     %td                   [sl] start date
sl_end          int     %td                   [sl] end date
sl_dur          byte    %9.0g                 [sl] duration
```

# Macro functions

## Exercise I

**Remarks to Exercise I:**

- Note that we use a loop here, but the code works also if you write all lines inside the curly braces `{ }` for every `sl_` var separately.
- First, we store the old variable label in the local `oldlab`.
- Next, we use the `substr` command to cut the `[sick leave]` prefix out of the variable labels. Here we use a little trick: Instead of specifying that the label part always starts at the 13th position in the string, we tell Stata to look for the position of `]` in the string, and then specify that the actual label starts 2 digits after `]`. This code is generalizable if you have different prefixes.
- The rest of the code is straight forward. Define another local `newlab` which adds the new prefix to the label snippet extracted before, and then attach the new label to the var.

# Macro functions

## Exercise II

**Exercise: Using macro functions, part 2**

Write a code that automatically opens all datasets in your data folder and saves them to use with older Stata versions using the `saveold` command.[a]

[a] Stata is backward compatible but not fully forward compatible, so you should always use `saveold` if you know that coworkers use older Stata versions.

# Macro functions

## Exercise II

```
. local datasets : dir "data" files "*.dta"

. di `datasets´
pdmv_sl.dta

.
. foreach data of local datasets {
  2.         use data/`data´, clear
  3.         saveold data/`data´, replace v(11)
  4. }
(All sick leaves 2004-2012 for 10% sample of Austrian employees)
(saving in Stata 12 format, which Stata 11 can read)
file data/pdmv_sl.dta saved
```

# B.5

*Loops*

# `foreach` and `forvalues`

- Loops ($\Longrightarrow$ programs that repeat the same command for several variables without having to write the same code for all variables) in Stata are typically done with the `foreach` and `forvalues` commands.

- A simple **numeric loop** for a defined and finite set of values over which to iterate can be accomplished using `forvalues`:

```
forval i = 1/5 {
   g p_educ‘i’ = p_educ == ‘i’
}
```

  Here, we define a local macro `i` as the **loop index.** Following an $=$ we give a range of values (in form of a *numlist*) which `i` is going to take on.

- You can write as many commands as you want inside the loop. Note also that you can nest multiple loops if you want to iterate over more than one var.

# `foreach` and `forvalues`

- With `foreach` you can loop over **any set of items,** regardless of pattern.

- There are many possible applications for `foreach` loops (some of which we have already learned in this module), but its single most important feature is that it allows to loop over variables:

```
foreach v in p_age p_female p_educ {
   sum `v'
   corr sl_dur `v'
}
```

  Here we define a local `v` as the variable (or *loop index*) to iterate over. The loop will compute summary statistics and correlation coefficients for the three worker vars in the data.

- Foreach allows different list types, instead of `in` *`list`* you may write
  - `of local` *`localmacroname`*
  - `of global` *`globalmacroname`*
  - `of varlist` *`varlist`*

# B.6

## *Scalars and matrices*

# Scalars

- Aside from macros, there are two more object containers in Stata that have certain properties and serve different purposes: **scalars and matrices.**
- Scalars can hold numeric and string values as well, but — unlike macros — can hold only **one** single value.
- Most estimation and data analysis commands return crucial parameters as scalars. They can be accessed via `return list` and are also summarized in the respective help files.
- A scalar can be addressed by simply naming it (macros have to be *dereferenced*), similar to variables in Stata. Avoid using the same names!

# Scalars

## Exercise

**Exercise: Manually compute 95% CI**

In Stata you often have to calculate confidence intervals yourself, because commands such as `collapse` do not allow to compute them directly. Obtain the mean and standard deviation of the variable `sl_dur` using the `summarize` command, and use only ingredients from its `return list` to compute the 95% confidence interval of the population mean. Use `ci mean sl_dur` to validate your result.

# Scalars

## Exercise

```
. su sl_dur
```

| Variable | Obs | Mean | Std. Dev. | Min | Max |
|---|---|---|---|---|---|
| sl_dur | 322,375 | 6.020889 | 5.386769 | 1 | 44 |

```
. return list, all

scalars:
                  r(N) =  322375
              r(sum_w) =  322375
             r(mean) =  6.020888716556805
              r(Var) =  29.01728221067055
               r(sd) =  5.386769181120586
             r(min) =  1
             r(max) =  44
             r(sum) =  1940984

. sca se = r(sd)/sqrt(r(N))

. sca sl_lb = r(mean) - invnormal(0.975)*se

. sca sl_ub = r(mean) + invnormal(0.975)*se

. di "95% CI lower bound: " sl_lb ", upper bound: " sl_ub
95% CI lower bound: 6.0022937, upper bound: 6.0394837

. ci means sl_dur
```

| Variable | Obs | Mean | Std. Err. | [95% Conf. | Interval] |
|---|---|---|---|---|---|
| sl_dur | 322,375 | 6.020889 | .0094874 | 6.002294 | 6.039484 |

# Matrices

- Stata provides a broad range on matrix operations for real matrices (and even a dedicated matrix language which we may encounter in the last lecture).
  - `help matrix` [Manual link: Summary of matrix commands]
- Estimation commands such as `reg` store coefficient matrices in `e(b)` and variance-covariance matrices in `e(V)`. Additionally, test statistics are often saved in matrices and can be addressed as such.
  - `matrix` *X* `=` *mname*
  - `matrix list` *X*
- Matrix commands are often useful for housekeeping; e.g., creating non-standard tables for export. We will discuss specific applications as we go along.