Statistisk sentralbyrå
Statistics Norway

Data processing in Sas, Spss, Stata, R and Python. A comparison

Kristian Lønø

TALL SOM FORTELLER

NOTATER / DOCUMENTS

2023/1

In the series Documents, documentation, method descriptions, model descriptions and standards are published.

Published: 13 January 2023

ISBN 978-82-587-1659-1 (electronic)
ISSN 2535-7271 (electronic)

Symbols in tables	Symbol
Category not applicable Figures do not exist at this time, because the category was not in use when the figures were collected.	.
Not available Figures have not been entered into our databases or are too unreliable to be published.	..
Confidential Figures are not published to avoid identifying persons or companies.	:
Decimal punctuation mark	.

Preface

When we are working with Statistical data we use software programs for data processing, analysis and tabulation. Which software to choose is depending on different factors like financial matters, management decisions, staff requests and so on. Five of the most commonly used software packages are the commercial Sas, Spss and Stata and the non-commercial R and Python.

This document gives a brief comparison between these software packages on how to do basic data processing for statistical surveys. It is meant to help employees who know one of the packages to learn some basics of the other ones. This is needed if the company changes from one software to another. It will also be useful for staff who co-operates with other companies who use other software than he or she usually works with. We can also use it as an introduction to one or more of the different softwares.

The versions used of the different software for this document are:

• Sas	9.4 M6
• Spss	27.0.1.0
• Stata	16.0
• R	4.0.0
• Python	3.10.5

As software always develop some of the program examples may be outdated when new versions arrive.

The author would like to thank Christian Thindberg and Anne-Marte Krogsrud for valuable comments.

Statistics Norway, 30 November 2022

Christian Thindberg

Contents

1. Introduction

The 3 commercial software packages Sas, Spss and Stata are all developed and maintained by American companies. They all have license models where we pay a yearly fee for the licenses. However, Stata also has an option to pay for a version of the package once and then use it forever. If we then want a new version of Stata, we must pay for it. The non-commercial package R can be downloaded free from the Internet, for instance from this site: https://cran.r-project.org/bin/windows/base/. It is also useful to download the R user interface RStudio, which can be found here: https://rstudio.com/products/rstudio/download/. The Python software is also free. When we use Python, we usually have a user interface as well. Both Python and a user interface may be downloaded from the Anaconda web site: https://www.anaconda.com/products/distribution.

They all have a user interface with different windows and pull-down menus. All use programming code that can either be written or generated through menus or wizards. Spss and Stata call their coding sets of commands, while Sas uses sets of statements. R also uses commands, although the commands are actually functions. Written or generated code should be stored with filename extensions according to their defaults:

• Sas	.sas
• Spss	.sps
• Stata	.do or .ado
• R	.R
• Python	.py (or .ipynb for Jupyter Notebooks)

Each package comes with lots of modules or packages as they are called in R. For Sas the primary need will be Sas Foundation, also called Base Sas, and Enterprise Guide. In addition, the module “Access to PC file formats” can be useful if easy data interchange with Spss and Stata is needed. The basics in Spss are in their Statistics Standard Edition. Stata operates with different modules depending on the size of the data used and the complexity in the usage. Stata/IC (the smallest one) will be suitable for most users. Whenever we need another package in R or Python we can download from the Internet and install it.

To install the software, follow the instructions given when you received it.

This document is based on version Sas 9.4, Sas Enterprise guide 8.3, Spss 27 and Stata 16, R 4.0.0, Python 3.10.5, but most of the functionality described will be available in previous versions.

For more info these are their web sites:
https://www.sas.com/
https://www.ibm.com/analytics/spss-statistics-software
https://www.stata.com/
https://cran.r-project.org/
https://www.python.org/

2. The user interfaces

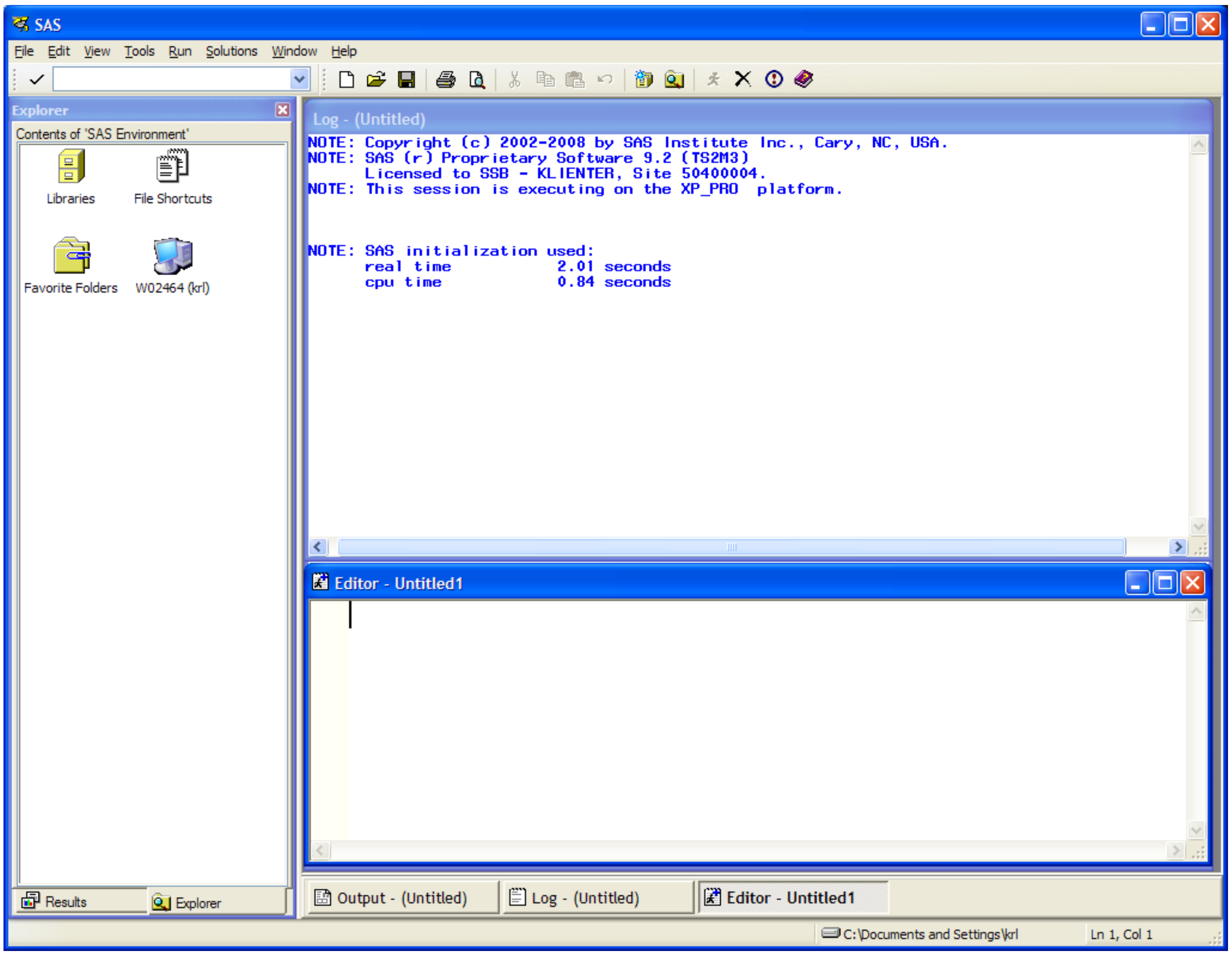
The user interface is an important part of every software. They differ, and it may be confusing to understand the user interface when we are used to one software package and will learn another one. We will now look at the basics of the user interfaces.

2.1. Sas

Base Sas

There are four important windows in Sas. They are usually opened when Sas is started:

• Explorer	Local Sas explorer, for looking at data
• Editor	Editor for writing our own programs
• Log	Execution log
• Output	Listings



```
NOTE: Copyright (c) 2002-2008 by SAS Institute Inc., Cary, NC, USA.
NOTE: SAS (r) Proprietary Software 9.2 (TS2M3)
      Licensed to SSB - KLIENTER, Site 50400004.
NOTE: This session is executing on the XP_PRO  platform.



NOTE: SAS initialization used:
      real time           2.01 seconds
      cpu time            0.84 seconds
```

In base Sas we must write most our programs ourselves. There are a few menus which we can use to generate statements, however we mostly have to write the statements.

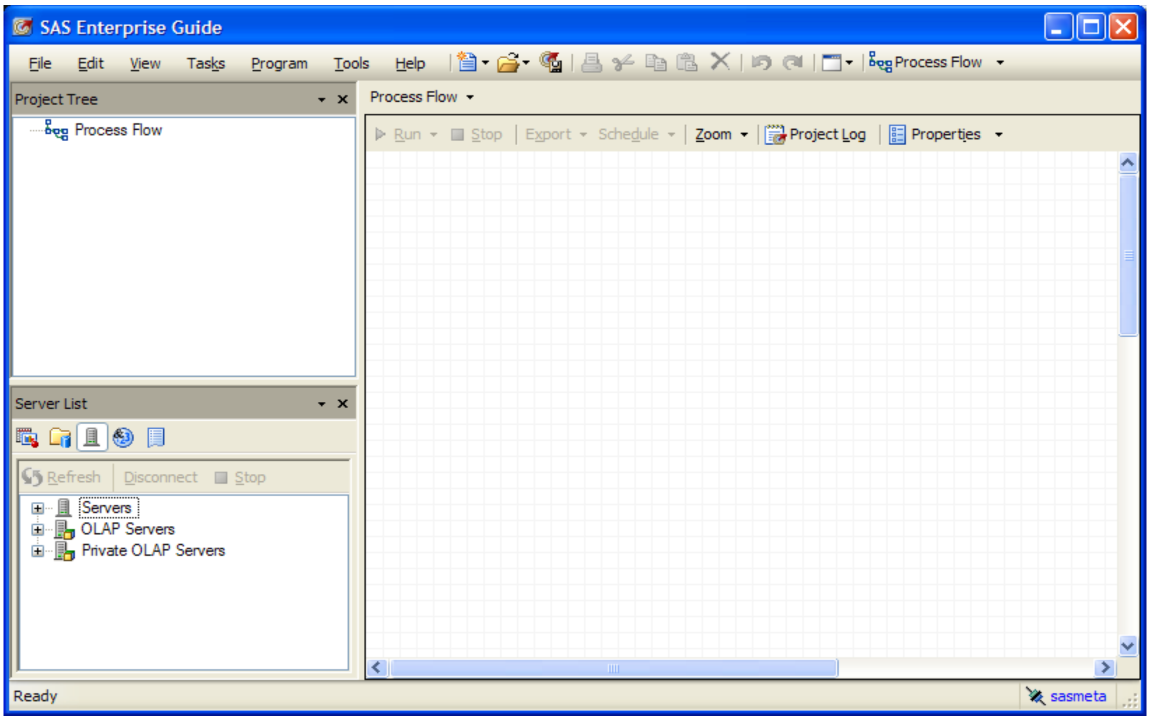
Sas Enterprise guide

Sas has a module called Enterprise Guide which provides menus for generating programs. It also gives us a flow chart of our processes. If Enterprise guide is included in your Sas license it will be the preferred user interface.

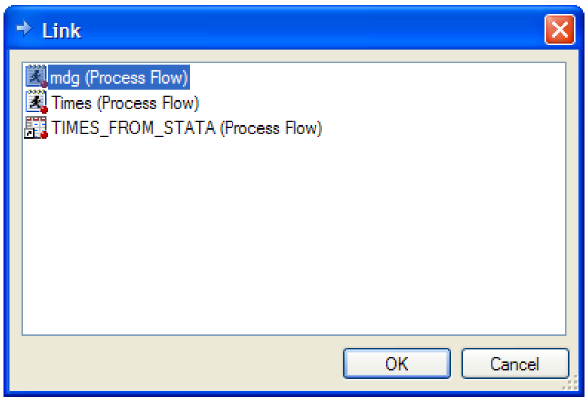
The main user interface in Enterprise Guide (EG) is the Process flow diagram. Every task (program) we create will be added here as a separate icon. For every task there will be different windows in separate tabs. These tabs may include these windows:

• Program	Written syntax
• Code	Generated syntax
• Log	Program log with notes, warnings, and error messages
• Output	Browsing of output dataset(s)
• Results	Listings
• Input Data	Browsing of input dataset

There is a menu bar where we can use menus or wizards to generate code and execute programs.



To open an editor window where we write syntax, we use the menu File > New > Program. Each of our programs will have an icon in the process flow. If we use the tasks in the Tasks menu to create syntax, Sas will add the arrows between the tasks in the process flow. When we write our own programs, we must do this linkage ourselves. We can do this by right-clicking on an icon in the process-flow and then choose Link. A menu with the possible links will appear and we choose the right one:



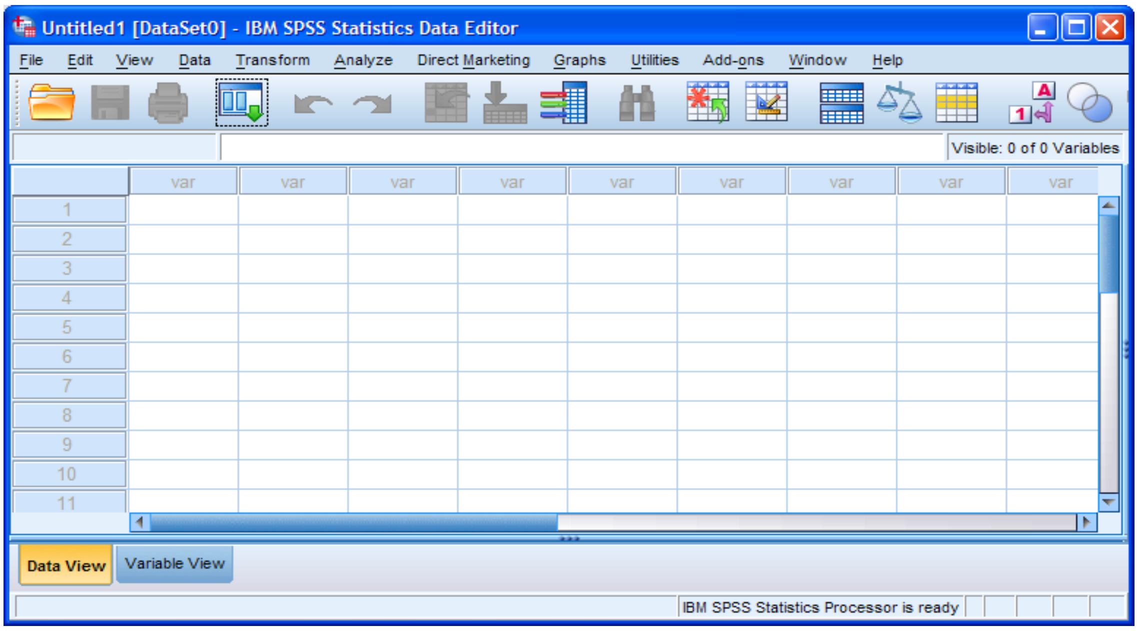
2.2. Spss

Spss has three important windows, of which the first two of these are usually opened when Spss is started:

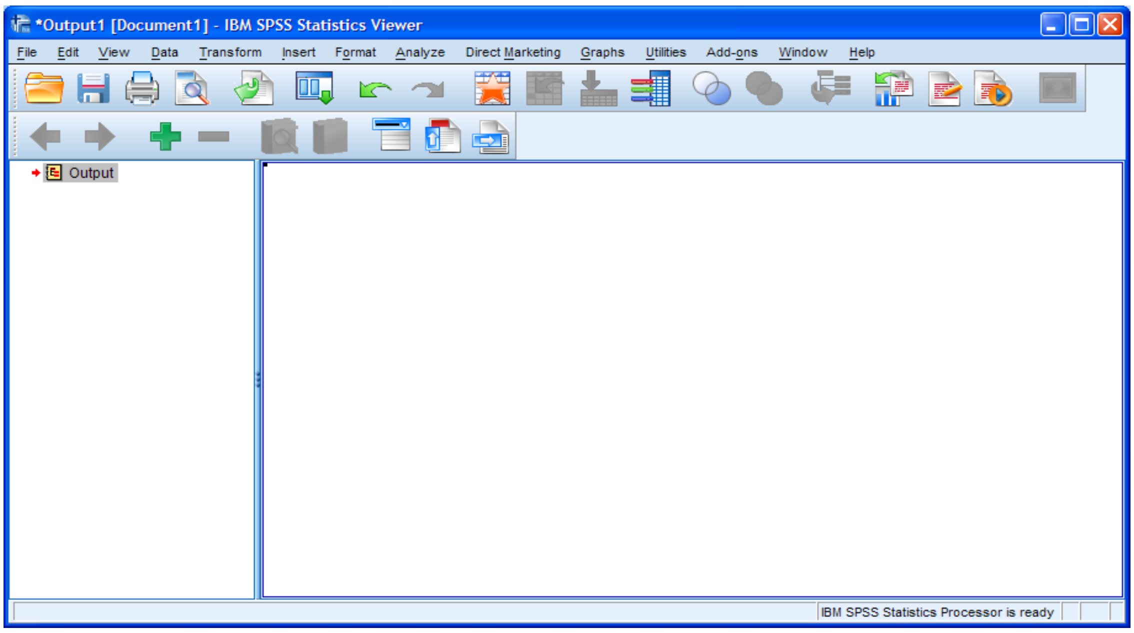
• Data Editor	Browsing and editing of dataset in use
• Viewer	Listings, both output, commands, and log
• Syntax Editor	For writing commands ourselves

These are opened in separate windows. If the syntax editor is not opened, we can open it with File, New, Syntax.

The Data editor window has two tabs, the Data and the Variable view. Here is the Data view:


Untitled1 [DataSet0] - IBM SPSS Statistics Data Editor
File Edit View Data Transform Analyze Direct Marketing Graphs Utilities Add-ons Window Help
Visible: 0 of 0 Variables
var
Data View
Variable View
IBM SPSS Statistics Processor is ready

The Output window when Spss is opened, all logs and listings will be written to this window:

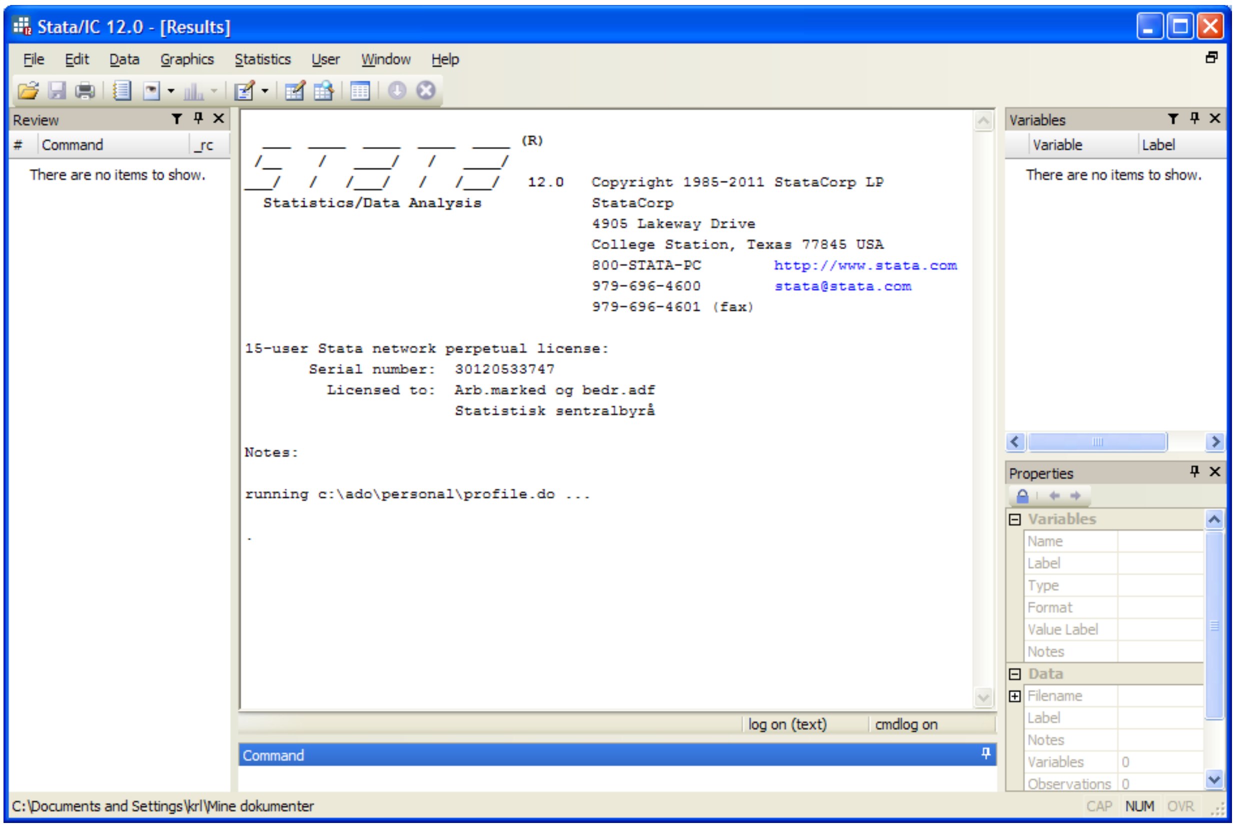

*Output1 [Document1] - IBM SPSS Statistics Viewer
File Edit View Data Transform Insert Format Analyze Direct Marketing Graphs Utilities Add-ons Window Help
Output
IBM SPSS Statistics Processor is ready

Spss has several menus which can be used to generate syntax. However, to generate syntax is not the default for these menus. The default is to just run the program and show the result in the Viewer window. To save the syntax we must tell Spss to paste the syntax into the Syntax editor. Then we can run the pasted syntax. It is always wise to save the syntax as it makes it possible to re-run programs and it will also document the process.

2.3. Stata

The user interface is divided into these windows:

• Review	List of executed commands
• Variables	Variables available for use
• Properties	Attributes for variables available for use
• Command	For writing single line commands (executed when *Enter* is pressed)
• Log	Log and listings viewer



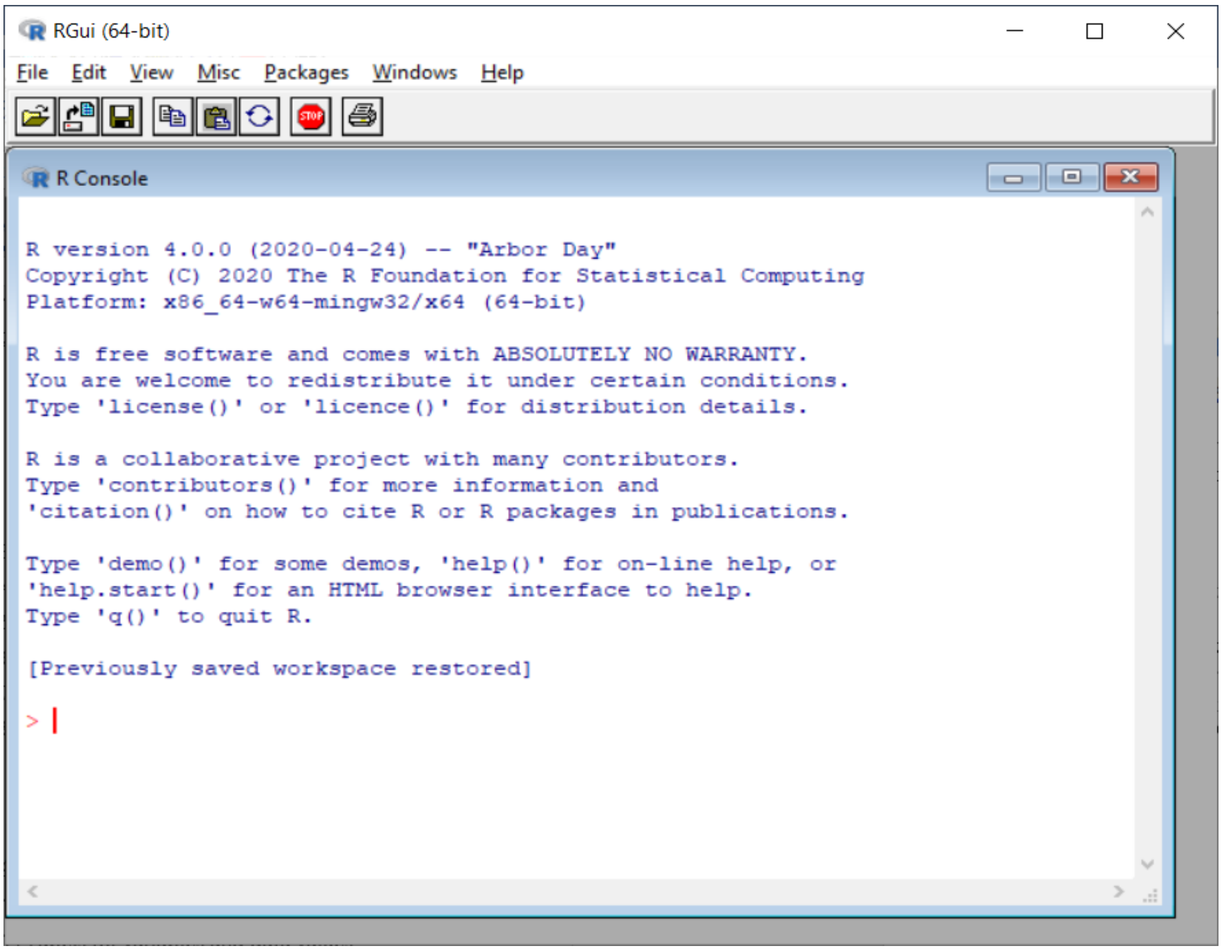
```
  ___  ____  ____  ____  ____ (R)
 /__    /   ____/   /   ____/
___/   /   /___/   /   /___/   12.0   Copyright 1985-2011 StataCorp LP
  Statistics/Data Analysis            StataCorp
                                      4905 Lakeway Drive
                                      College Station, Texas 77845 USA
                                      800-STATA-PC        http://www.stata.com
                                      979-696-4600        stata@stata.com
                                      979-696-4601 (fax)

15-user Stata network perpetual license:
       Serial number:  30120533747
         Licensed to:  Arb.marked og bedr.adf
                       Statistisk sentralbyrå

Notes:

running c:\ado\personal\profile.do ...

.
```

In addition, these windows are important:

• Do-file editor	For writing commands (opens from the Window menu or Ctrl+9)
• Data editor	For browsing and editing data (opens from the Window menu or Ctrl+8)

2.4. R

The default R interface is quite simple:

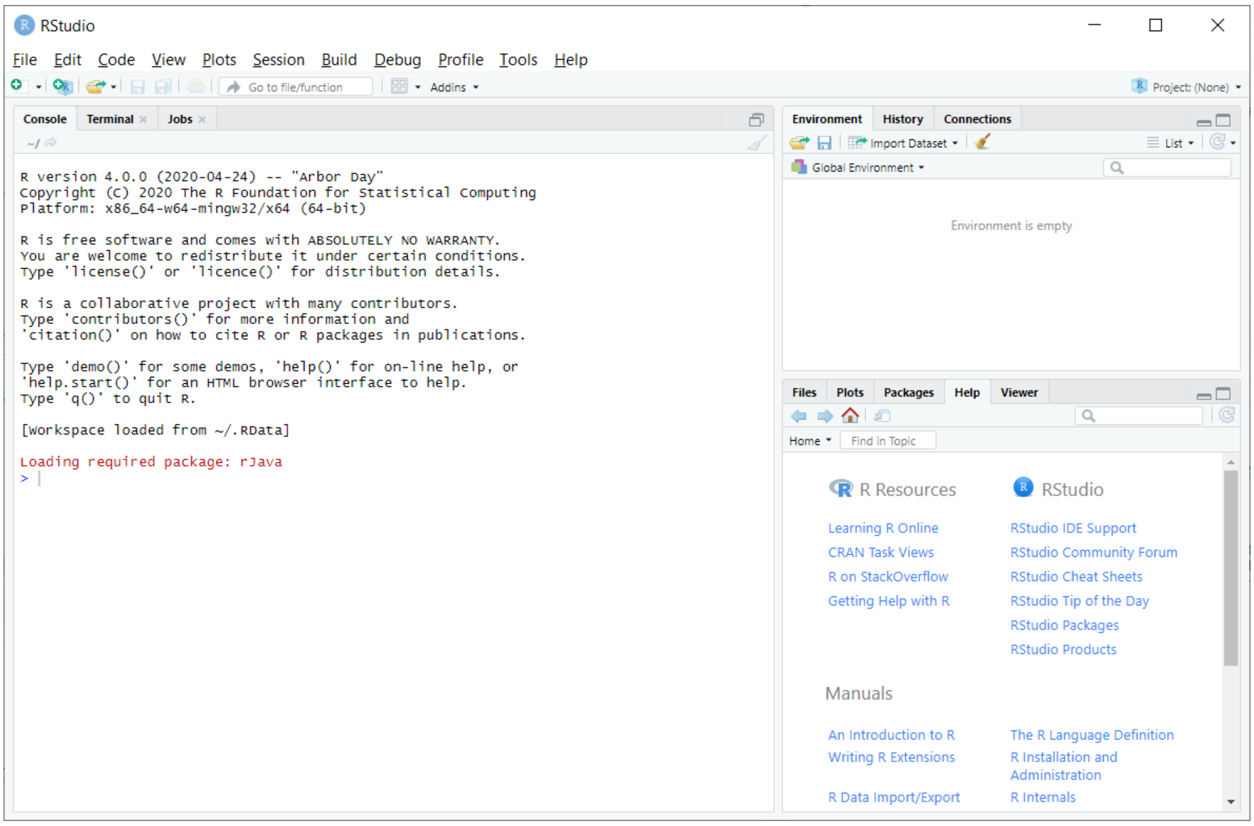


```
R version 4.0.0 (2020-04-24) -- "Arbor Day"
Copyright (C) 2020 The R Foundation for Statistical Computing
Platform: x86_64-w64-mingw32/x64 (64-bit)

R is free software and comes with ABSOLUTELY NO WARRANTY.
You are welcome to redistribute it under certain conditions.
Type 'license()' or 'licence()' for distribution details.

R is a collaborative project with many contributors.
Type 'contributors()' for more information and
'citation()' on how to cite R or R packages in publications.

Type 'demo()' for some demos, 'help()' for on-line help, or
'help.start()' for an HTML browser interface to help.
Type 'q()' to quit R.

[Previously saved workspace restored]

> |
```

It is so simple that it is recommended to use the RStudio interface instead when working in R, so we will concentrate on that.

The RStudio interface is divided into these windows at start-up:

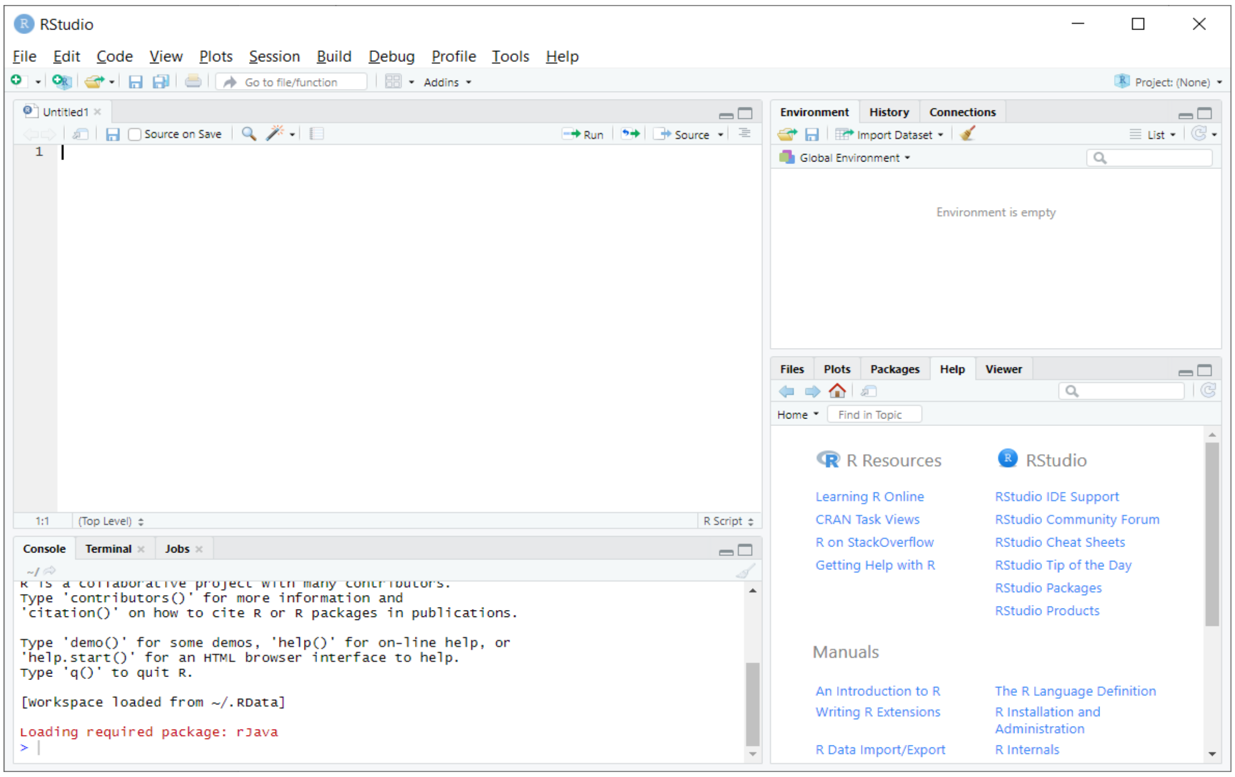
- Console/Terminal/Jobs
- Environment/History/Connections
- Files/Plots/Packages/Help/Viewer



```
R version 4.0.0 (2020-04-24) -- "Arbor Day"
Copyright (C) 2020 The R Foundation for Statistical Computing
Platform: x86_64-w64-mingw32/x64 (64-bit)

R is free software and comes with ABSOLUTELY NO WARRANTY.
You are welcome to redistribute it under certain conditions.
Type 'license()' or 'licence()' for distribution details.

R is a collaborative project with many contributors.
Type 'contributors()' for more information and
'citation()' on how to cite R or R packages in publications.

Type 'demo()' for some demos, 'help()' for on-line help, or
'help.start()' for an HTML browser interface to help.
Type 'q()' to quit R.

[Workspace loaded from ~/.RData]

Loading required package: rJava
> |
```

In the Console window our executed commands and lists will appear. Error messages will also appear here. In the Help window we find help to the different functions (commands). When we make Plots they will show up in the Plot window. The environment window will contain all data that is in the memory.

In R we will usually write scripts and to do that we will open an empty program editor with File, New File, RScript (CTRL+SHIFT+N). The RStudio Editor will now open with an empty script:

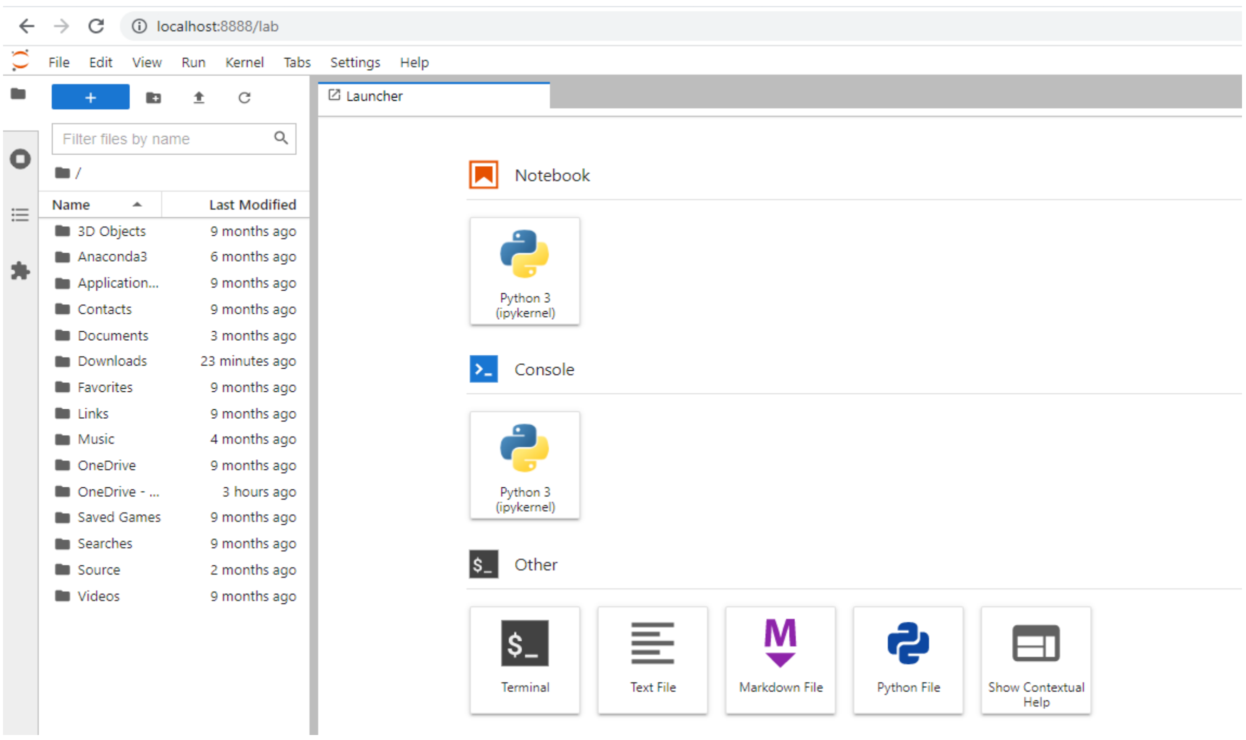


```
R is a collaborative project with many contributors.
Type 'contributors()' for more information and
'citation()' on how to cite R or R packages in publications.

Type 'demo()' for some demos, 'help()' for on-line help, or
'help.start()' for an HTML browser interface to help.
Type 'q()' to quit R.

[Workspace loaded from ~/.RData]

Loading required package: rJava
> |
```

Another important window is the data viewer, which opens a data frame. With this, we have an easy and flexible way to look at our data frames. We use the *View* command to open a data frame in the data viewer.

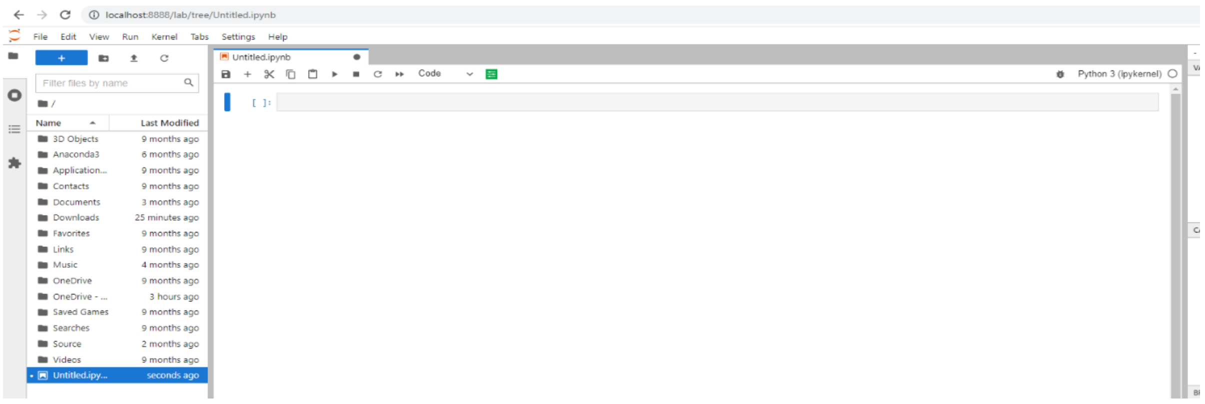
2.5. Python

We can use different interfaces when working with Python. In this document we will use JupyterLab, which can be downloaded from the Anaconda web site: https://www.anaconda.com/products/distribution. After it is installed, we open the Anaconda Navigator and from there we launch JupyterLab. This will open a new page in a web browser that will look something like this:



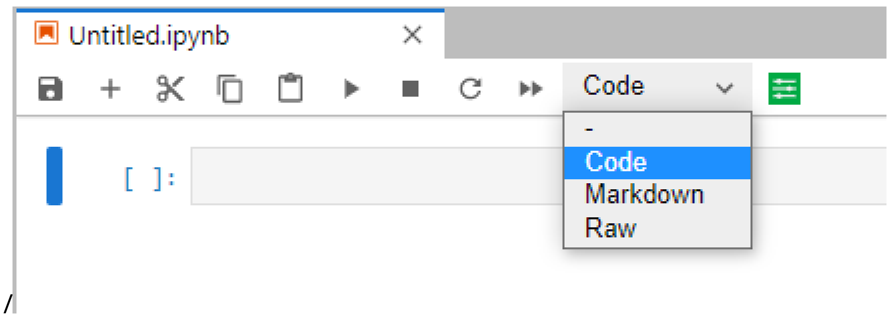
Name	Last Modified
3D Objects	9 months ago
Anaconda3	6 months ago
Application...	9 months ago
Contacts	9 months ago
Documents	3 months ago
Downloads	23 minutes ago
Favorites	9 months ago
Links	9 months ago
Music	4 months ago
OneDrive	9 months ago
OneDrive - ...	3 hours ago
Saved Games	9 months ago
Searches	9 months ago
Source	2 months ago
Videos	9 months ago

We open a new notebook by clicking on the icon below Notebook and an empty notebook will be opened:


localhost:8888/lab/tree/Untitled.ipynb
File Edit View Run Kernel Tabs Settings Help
Filter files by name
Untitled.ipynb
Code
Python 3 (ipykernel)
[ ]:

Name	Last Modified
3D Objects	9 months ago
Anaconda3	6 months ago
Application...	9 months ago
Contacts	9 months ago
Documents	3 months ago
Downloads	25 minutes ago
Favorites	9 months ago
Links	9 months ago
Music	4 months ago
OneDrive	9 months ago
OneDrive - ...	3 hours ago
Saved Games	9 months ago
Searches	9 months ago
Source	2 months ago
Videos	9 months ago
Untitled.ipy...	seconds ago

The notebook open with one code cell. Each cell can be either Python code, a Markdown text or raw text. We mostly use the Markdown cells for documentation. The code cells will contain our Python code. To choose cell type click on the code text above the cells:



In Jupyter notebooks we operate in two different modes:

- Edit To type into a cell like in a normal editor. In this mode we write our code and comments.
- Command To operate on whole cells, for instance to run, move or delete them.

We swap from code to command with ESC. From command to code we swap with Enter

To run a program, we can click on the Run icon at the top of the window (▶). Or we can use these key combinations:

Shift+Enter	Run the program in the cell and move to next cell
Ctrl+Enter	Run the program and stay in the same cell
Alt+Enter	Run the program in the cell and add a new cell below

The code and markdown cells are input cells. This means we can write code or documentation within them. We also have output cells. These cells show output from our Python program code.

3. Naming conventions

The naming conventions for variables are a little bit different between the packages. These variable naming conventions could well be used for dataset names and other naming. Here are the most important naming rules:

Rule	**Sas**	**Spss**	**Stata**	**R**	**Python**
Max length	32	64	32	Only limited by the system resources available	Only limited by the system resources available
Allowed letters	A-Z, a-z	Any letter allowed	A-Z, a-z	Any letter allowed	Any letter allowed
Case sensitive?	No	No	Yes	Yes	Yes
Special characters	_ allowed	_ . and non-punctuation characters allowed (. is usually not recommended)	_ allowed, but if put in the first position it indicates a macro variable name	_ . are allowed. Must start with a letter or period. If it starts with a period, it cannot be followed by a digit.	_ allowed
Space	Not allowed	Not allowed	Not allowed	Allowed, but not recommended	Not allowed
Numbers	Allowed, but not in the first position	Allowed, but not in the first position	Allowed, but not in the first position	Allowed, but not in the first position	Allowed, but not in the first position
$	Allowed in the first position to refer to a character format name	Allowed, but in the first position reserved for system variables	Not allowed	Allowed, but not recommended as it is used to distinguish between dataset name and variable name	Not allowed
#	Not allowed	Allowed, but in the first position reserved for scratch variables	Not allowed	Allowed, but not recommended	Not allowed
&	Allowed in the first position to refer to a macro variable	Not allowed	Not allowed	Allowed, but not recommended	Not allowed
!	Not allowed	Allowed in the first position to refer to a macro variable	Not allowed	Allowed, but not recommended	Not allowed
@	Not allowed	Allowed	Not allowed	Allowed, but not recommended	Not allowed

As we sometimes interchange datasets between software packages it is smart to name variables, so they are valid for all of these packages at the same time. In Sas and Spss it is a convention to use capital letters for command names and otherwise use low case letters. As Stata, R and Python are case-sensitive we must use the commands as they are described, mostly lowercase. This goes for Stata, R and Python variable names as well; a variable called b2 is not the same variable as B2. In Sas and Spss the variable b2 is the same variable as B2. In labels and titles, we are free to use both upper and lower case as we like. Sas has a notation that can allow us use any text 32 character or less in length as a name ('name'n). However, it is recommended to stick to the name conventions.

4. Operators

We always have to use operators when we are working with programming software. Here are the most used ones and how they are written:

Operator	**Sas**	**Spss**	**Stata**	**R**	**Python**
Arithmetic:					
Addition	+	+	+	+	+
Subtraction	-	-	-	-	-
Multiplication	*	*	*	*	*
Division	/	/	/	/	/
Exponentiation	**	**	^	** (or ^)	**
Equal (set to)	= (or eq)	= (or eq)	=	=	=
Relational:					
Equal (check for)	= (or eq)	= (or eq)	==	==	==
Not equal	ne (or ~= or ^= or <>)	ne (or ~= or <>)	~= (or !=)	!=	!=
Less than	< (or lt)	< (or lt)	<	<	<
Less than or equal	<= (or le)	<= (or le)	<=	<=	<=
Greater than	> (or gt)	> (or gt)	>	>	>
Greater than or equal	>= (or ge)	>= (or ge)	>=	>=	>=
Logical:					
Reverse the expression	not (or ^ or ~)	not (or ~)	! (or ~)	!	~
Both relations true	and (or &)	and (or &)	&	&	and
Either relation true	or (or |)	or (or |)	|	|	or

We can change the order of evaluation by using parentheses. It is always wise to use parentheses to make sure that our expressions evaluate in the order we want.

4.1. Sas

The order of evaluation is depending on which group the operators belong to. The priority of evaluation is:

Group	**Description**	**Associativity**
1	**, + (as prefix), - (as prefix), NOT	right to left
2	*, /	left to right
3	+ (addition), - (subtraction)	left to right
4	<, <, =, NE, >, >=.	left to right
5	and	left to right
6	or	left to right

4.2. Spss

Order of evaluation
Not
Exponentiation
Multiplication
Division
Addition and subtraction
Ne
>, <, >=, <=
==
And
Or

4.3. Stata

Order of evaluation
!
^
- (negation)
/
*
- (subtraction), + (addition)
and
!= (or ~=)
>, <, <=, >=
==
&
|

4.4. R

The operators are evaluated in the order shown below:

Operator	**Description**	**Associativity**
^	Exponent	Right to Left
-x, +x	Unary minus, Unary plus	Left to Right
%%	Modulus	Left to Right
*, /	Multiplication, Division	Left to Right
+, –	Addition, Subtraction	Left to Right
<, >, <=, >=, ==, !=	Comparisions	Left to Right
!	Logical NOT	Left to Right
&, &&	Logical AND	Left to Right
|, ||	Logical OR	Left to Right
->, ->>	Rightward assignment	Left to Right
<-, <<-	Leftward assignment	Right to Left
=	Leftward assignment	Right to Left

4.5. Python

The operators are evaluated in a defined order (highest to lowest):

Operator	**Name**
(...), [...], {...}	Tuple, list, and dictionary creation
´...´	String conversion
s[i], s[i:j]	Indexing and slicing
s.attr	Attributes
f(...)	Function calls
+x, -x, ~x	Unary operators
x ** y	Power (right associative)
x * y, x / y, x // y, x % y	Multiplication, division, floor division, modulo
x + y, x - y	Addition, subtraction
x << y, x >> y	Bit-shifting
x & y	Bitwise and
x ^ y	Bitwise exclusive or
x | y	Bitwise or
x < y, x <= y, x > y, x >= y, x == y, x != y, x <> y	Comparison tests
x is y, x is not y	Identity tests
x in s, x not in s	Sequence membership tests
not x	Logical negation
x and y	Logical and
x or y	Logical or
lambda args: expr	Anonymous function

5. Datasets

All packages except Python store data in their own proprietary format with a special extension in the file names. Python uses open-source formats, for instance those mentioned below:

• Sas:	sas7bdat
• Spss:	sav
• Stata:	dta
• R:	RData or rds
• Python:	pkl or parquet

Both Spss and Stata operate with active datasets. In Stata we can only have one dataset opened in the data editor at a time and this will be the dataset the commands will be executed on. Spss also have an active dataset for which commands will be executed. However, we can have several datasets open at the same time. This is not recommended because then we have to make sure which dataset is the active one and this can be confusing and it easy to execute on the wrong dataset. It is best to have only one dataset opened at the time. To ensure this we set an option: Edit > options, and in the General tab, tick Open only one dataset at a time. The datasets in Spss and Stata are usually referenced to by their whole physical names unless a *cd* command is used. If so, we may use the dataset name without the path given in the *cd* command.

Sas does not operate with active datasets. We do not use the physical names in the programs either, we use aliases. These aliases consist of a *libref* which reference to the physical directory, in which the data is stored, and the name of the dataset (without the extension). The two parts of the dataset name is separated with a dot. The *libref* for temporary datasets is called Work and is defined when Sas starts. For these temporary datasets we can omit the *libref*. *Librefs* for permanent datasets are defined by using a *Libname* statement, see page 62.

R save all data in memory until we close R or actively remove the data from memory. R data are stored in vectors with given names, and we use these names to reference the vectors. R has also introduced data frames which are vectors organized in a way that makes it possible to treat them as regular datasets (called data frames in R). We may use the *attach* command to be able to reference variables within a data frame without using the name. There is also possible to use the *with* function in R to avoid repeating the data frame names.

Python also save all data in memory until we close the Python kernel or actively remove the data from memory. All internal data are stored in objects which can be of different formats, like lists, dictionaries, tuples, sets, frozenset and Pandas data frames. We will mostly work with Pandas data frames in this document; however we will look a little bit at dictionaries as well.

All folders must be created before we can write any datasets to these folders. It is wise to have a separate folder for the datasets we want to store permanently. This is common for all the compared softwares in this document.

6. Execution

The syntaxes all consist of commands (statements). Each command usually starts with a keyword and ends with a special character. In Sas the statements end with a semicolon (;), in Spss a period (.) is used and Stata uses a line break. We can change the end character in Stata, but it is usually not recommended. If a command continues to a new line in Stata, we use /// as a continuation marker. In Sas, Spss and R we don't need a continuation marker. In Spss it is not allowed to have an empty line within a command.

As Spss and Stata always executes on the active dataset, we can run commands separately. For Spss some commands are executed immediately, and some commands wait for the *Execute* command to be executed. Stata and R executes commands immediately. Sas needs to put statements together in either a Data step or a Proc step. The Data step is for regular programming and the Proc step uses ready-made procedures. The Data step always starts with a *Data* statement and the Proc step with a *Proc* statement. A step in Sas ends usually with a *Run* statement, but sometimes *Quit* is used instead.

When a program or syntax is written or generated, we will execute it. We may execute the whole program or just parts of it. In Sas there is a *Submit* command which executes the program. In base Sas it is found under the *Run* menu. As it is a command which is very often used it should be defined to a hot-key. The hot-keys are defined under Tools > options > Keys.

In Sas Enterprise guide there are many ways to Submit a program. The easiest way is to use the F3 button. When pressed the program is submitted. If a part of the program is marked only that part is submitted. There are always different Run options available as well.

To execute syntax in Spss we can use the menu Run. Under the Run menu there are different choices. The most common are *All* and *Selection*. The hot-key Ctrl-R will execute the selected syntax.

In Stata a command will be executed when *Enter* is pressed when it is written in the Command window. If we write the commands in the Do-file Editor, we can execute them from the *Tools* menu. There is also the hot-key Ctrl+D which executes the marked text. If no text is marked all the commands in the Do-file will be executed with Ctrl+D.

To run a script in R, we mark the lines we want to run and press CTRL+ENTER. To run a separate command, we press CTRL+ENTER when the cursor is on a line within the command.

For Python programs in Jupyter notebooks, we run a program in a cell by using one of the combinations CTRL+ENTER, SHIFT+ENTER or ALT+ENTER.

In Python, we have a basic part which is available when we start a Python session. However, there are lots of other packages that may be used. These must be imported before we can use them. It is common practise to put these imports in the first code cell in our projects. To run the examples in this document, we need to run these imports:

```
import pandas as pd
import numpy as np
from io import StringIO
import datetime as dt
```

Pandas are used to be able to work with data frames, which is very common when we do data processing. Numpy is a package for numeric computations which is widely used. StringIO is very helpful when we want to read inserted text files. Datetime is a module for extracting and calculating on dates.

7. Import of files

As all packages store data in their own proprietary format, the first we usually do is to import data. It may be imported from different formats. The most common formats are delimiter separated files, fixed positions files and Excel spreadsheets. Sas Enterprise guide has a wizard (task) for importing these types of files. Base Sas does also have a wizard, however it does not take fixed positions files and to be able to import Excel spreadsheets we have to have a license for the module “Access to PC file formats”. Spss has menus for importing delimiter separated files and Excel spreadsheets, but not for fixed positions files. Stata has menus for importing all three file types. If we use menus or wizards to import data, it is important to save the syntax that is generated.

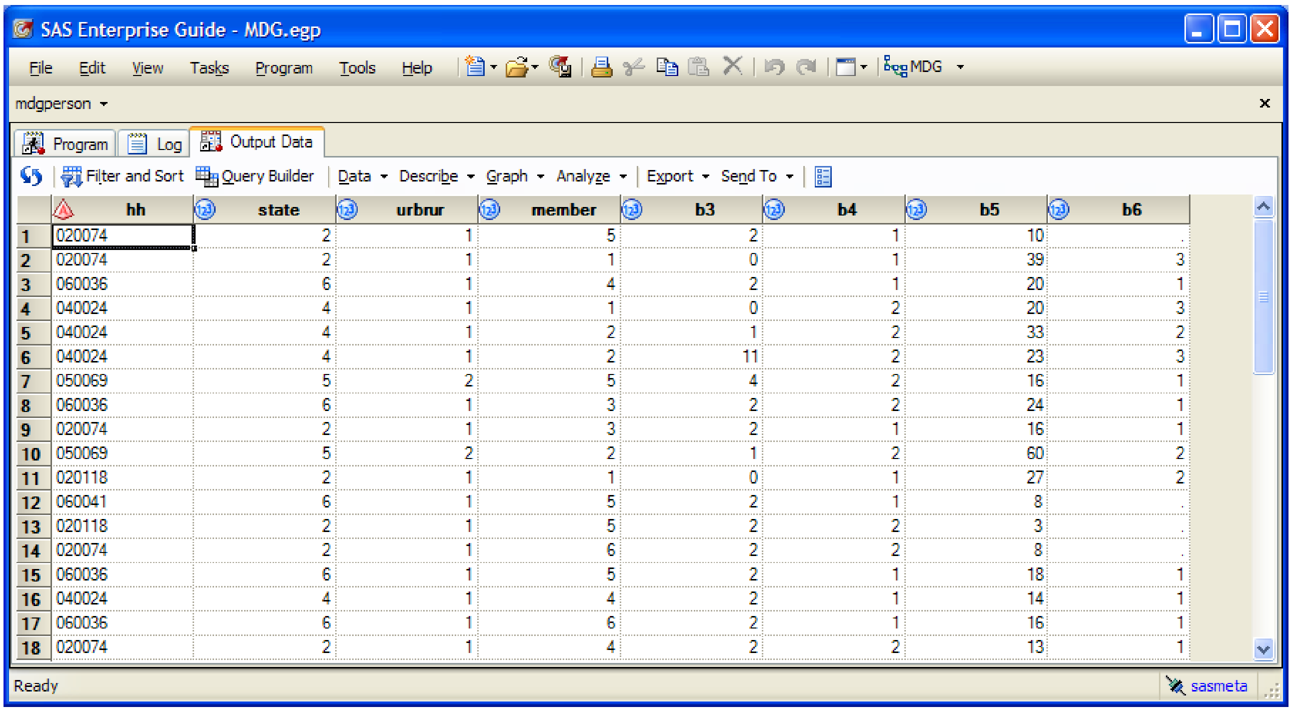
We will now look at how we use syntax to import a fixed positions file, the file is shown in the appendix on page 224.

7.1. Sas

```
DATA mdgperson ;
  INFILE 'H:\MDG\Data\mdgperson.txt' TRUNCOVER LRECL=15;
  INPUT
    @01 hh      $CHAR6.
    @07 state   1.
    @08 urbrur  1.
    @09 member  1.
    @10 b3      2.
    @12 b4      1.
    @13 b5      2.
    @15 b6      1.
    ;
RUN;
```

The *Data* statement starts the Data step and names the output dataset. The *Infile* statement names the file to be imported and the *Input* statement describes the different columns to be read. It is also used to name the variables in Sas and to decide which variables will be character ($char) and which will be numeric. The start position is given after the @ sign and the length is given as an informat after the variable name. Informats end with a dot (or a decimal number).

After we run this program the output data window looks like this:


SAS Enterprise Guide - MDG.egp
File Edit View Tasks Program Tools Help
MDG
mdgperson
Program Log Output Data
Filter and Sort Query Builder Data Describe Graph Analyze Export Send To
Ready
sasmeta

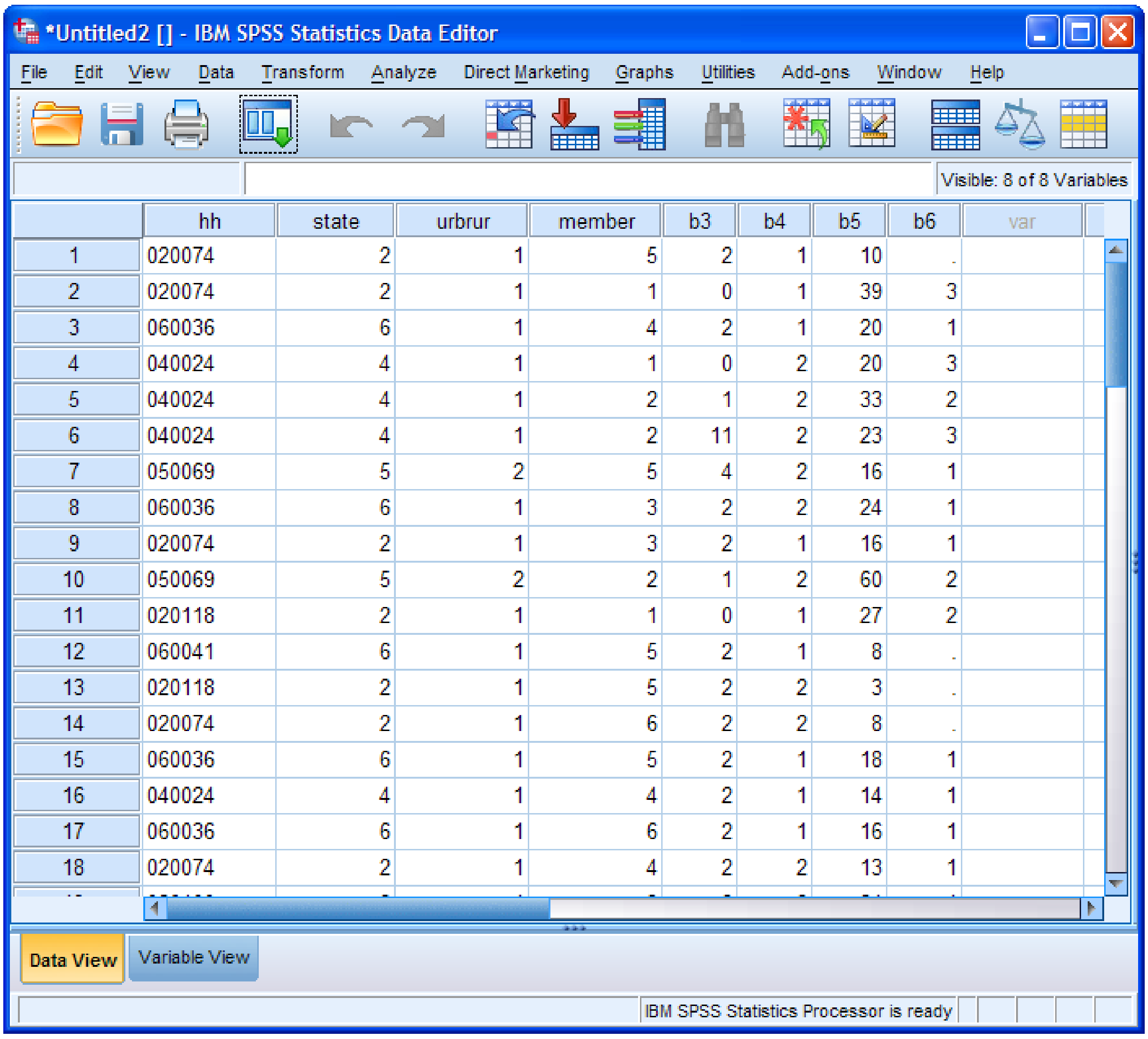
	hh	state	urbrur	member	b3	b4	b5	b6
1	020074	2	1	5	2	1	10	.
2	020074	2	1	1	0	1	39	3
3	060036	6	1	4	2	1	20	1
4	040024	4	1	1	0	2	20	3
5	040024	4	1	2	1	2	33	2
6	040024	4	1	2	11	2	23	3
7	050069	5	2	5	4	2	16	1
8	060036	6	1	3	2	2	24	1
9	020074	2	1	3	2	1	16	1
10	050069	5	2	2	1	2	60	2
11	020118	2	1	1	0	1	27	2
12	060041	6	1	5	2	1	8	.
13	020118	2	1	5	2	2	3	.
14	020074	2	1	6	2	2	8	.
15	060036	6	1	5	2	1	18	1
16	040024	4	1	4	2	1	14	1
17	060036	6	1	6	2	1	16	1
18	020074	2	1	4	2	2	13	1

7.2. Spss

```
DATA LIST FILE = 'H:\MDG\Data\mdgperson.txt' /
    hh          1 -     6 (A)
    state       7 -     7
    urbrur      8 -     8
    member      9 -     9
    b3         10 -    11
    b4         12 -    12
    b5         13 -    14
    b6         15 -    15
    .
EXECUTE.
SAVE OUTFILE='H:\MDG\Data\mdgperson.sav'.
```

The import is done with one command, the *Data list* command. It gives the name of the import file in the *File* subcommand. Then the variable names for Spss and the start and end positions are given. We also decide whether a variable shall be character (A) or numeric. The *Execute* command is needed for the actual import to be done. The *Save* command store the dataset in Spss format

When the syntax is executed, the imported data will be shown in the Data editor window:


*Untitled2 [] - IBM SPSS Statistics Data Editor
File Edit View Data Transform Analyze Direct Marketing Graphs Utilities Add-ons Window Help
Visible: 8 of 8 Variables
Data View
Variable View
IBM SPSS Statistics Processor is ready

	hh	state	urbrur	member	b3	b4	b5	b6	var
1	020074	2	1	5	2	1	10	.	
2	020074	2	1	1	0	1	39	3	
3	060036	6	1	4	2	1	20	1	
4	040024	4	1	1	0	2	20	3	
5	040024	4	1	2	1	2	33	2	
6	040024	4	1	2	11	2	23	3	
7	050069	5	2	5	4	2	16	1	
8	060036	6	1	3	2	2	24	1	
9	020074	2	1	3	2	1	16	1	
10	050069	5	2	2	1	2	60	2	
11	020118	2	1	1	0	1	27	2	
12	060041	6	1	5	2	1	8	.	
13	020118	2	1	5	2	2	3	.	
14	020074	2	1	6	2	2	8	.	
15	060036	6	1	5	2	1	18	1	
16	040024	4	1	4	2	1	14	1	
17	060036	6	1	6	2	1	16	1	
18	020074	2	1	4	2	2	13	1	

It is divided into two parts: the Data view and the Variable view. The data is shown in the Data view and the metadata is in the Variable view. The metadata consists of variable names, lengths, types, variable labels, and value labels.

When there is invalid data for numeric fields, messages like this will appear in the log and the value is set to missing:

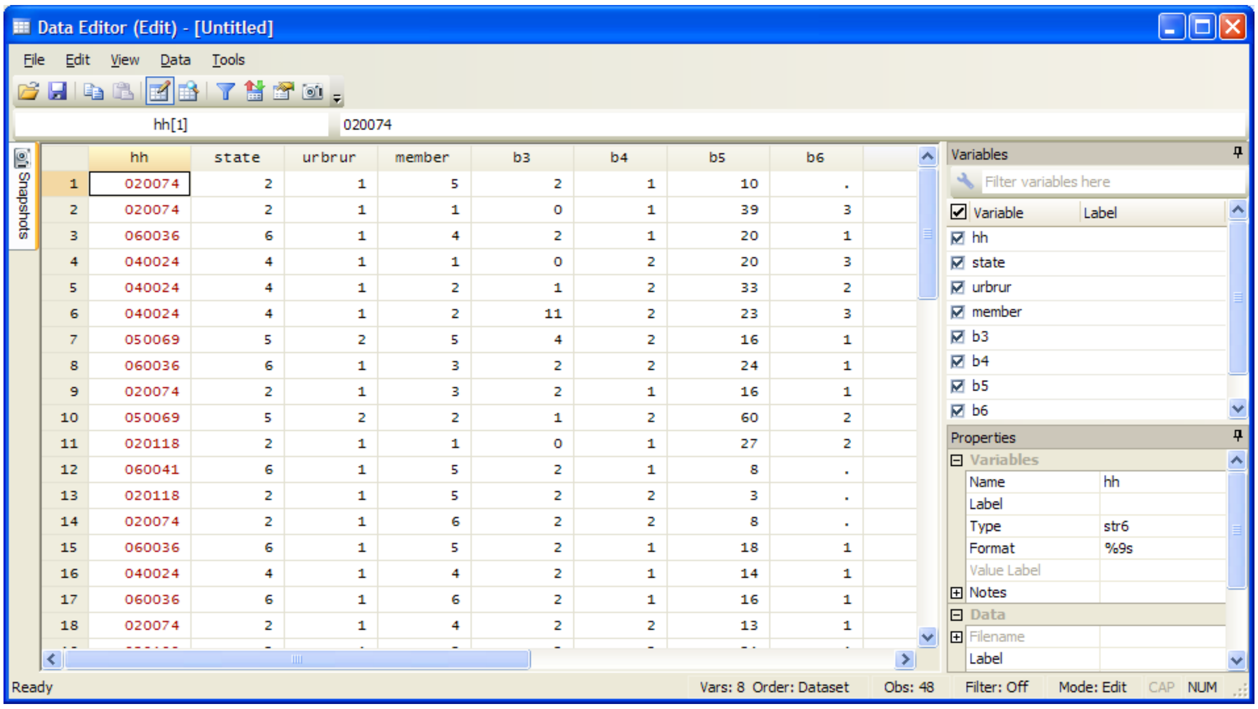
```
Warning # 1102

An invalid numeric field has been found.  The result has been set to the

system-missing value.

Command line: 50  Current case: 33  Current splitfile group: 1

Field contents: '.'

Record number: 33  Starting column: 15  Record length: 8192
```

7.3. Stata

```
infix                           ///
str6   hh       1 - 6           ///
       state    7 - 7           ///
       urbrur   8 - 8           ///
       member   9 - 9           ///
       b3       10 - 11         ///
       b4       12 - 12         ///
       b5       13 - 14         ///
       b6       15 - 15         ///
using h:\mdg\data\mdgperson.txt
save "h:\mdg\data\mdgperson.dta", replace
```

Stata also uses one command to import the file, *infix*. This command will use more than one line; hence we need to use the continuation marker ///. Character variables will be marked with the *str* format. The variable name, start and end positions for each variable is defined and the name of the import file is also needed. We save the dataset with the *save* command.

When Stata has executed the import program the data will not automatically be shown. To show the data we open the Data editor window (hot-key Ctrl+8):



hh[1] 020074

	hh	state	urbrur	member	b3	b4	b5	b6
1	020074	2	1	5	2	1	10	.
2	020074	2	1	1	0	1	39	3
3	060036	6	1	4	2	1	20	1
4	040024	4	1	1	0	2	20	3
5	040024	4	1	2	1	2	33	2
6	040024	4	1	2	11	2	23	3
7	050069	5	2	5	4	2	16	1
8	060036	6	1	3	2	2	24	1
9	020074	2	1	3	2	1	16	1
10	050069	5	2	2	1	2	60	2
11	020118	2	1	1	0	1	27	2
12	060041	6	1	5	2	1	8	.
13	020118	2	1	5	2	2	3	.
14	020074	2	1	6	2	2	8	.
15	060036	6	1	5	2	1	18	1
16	040024	4	1	4	2	1	14	1
17	060036	6	1	6	2	1	16	1
18	020074	2	1	4	2	2	13	1

Variable	Label
hh	
state	
urbrur	
member	
b3	
b4	
b5	
b6	

Variables	
Name	hh
Label	
Type	str6
Format	%9s
Value Label	
Notes	
Data	
Filename	
Label	

Ready   Vars: 8 Order: Dataset   Obs: 48   Filter: Off   Mode: Edit   CAP   NUM

If we want to re-run the import, we have to close the dataset first. This may be done with the *clear* command.

7.4. R

We use the read command to import datafiles. For fixed position files we use *read.fwf*. For delimited files we can use *read.csv* or *read.table*.

```
mdgperson <- read.fwf(
  file="H:/MDG/Data/mdgperson.txt",
  col.names = c("hh","state","urbrur","member","b3","b4","b5","b6"),
  widths=c(6, 1, 1, 1,  2, 1, 2, 1),

colClasses=c("character","character","character","numeric","numeric","numeric
","numeric","numeric"),
  skip=0,
  na.strings="."
  )
```

The *read.fwf* command will create a data frame by importing the file mentioned in the file argument. Beware that in R we must use slash (/) and not backslash (\) to separate folder from sub-folder in the file path. The data frame will be named mdgperson. It is assigned by using the <- operator which means that the result of the expression to the right of <- will be assigned to the name to the left of <-

We name the columns (or variables) in the data frame by using the *col.names* argument. The names are put into the concatenation function (c) in the same order as they appear in the data file. Furthermore, we must define the width and type of each column. We do that with the *widths* and *colClasses* arguments as seen above. As there are no space between each column in the data file, we tell R that by using the *skip* argument. Finally we use the *na.strings* argument to identify values for missing values, which in R is called not available (NA).

The data frame is not shown in R. However, we can use different commands to see the content of the data frame. We can use the *str* command (str is short for structure) like this:

```
str(mdgperson)
```

This will write an output like this to the console window:

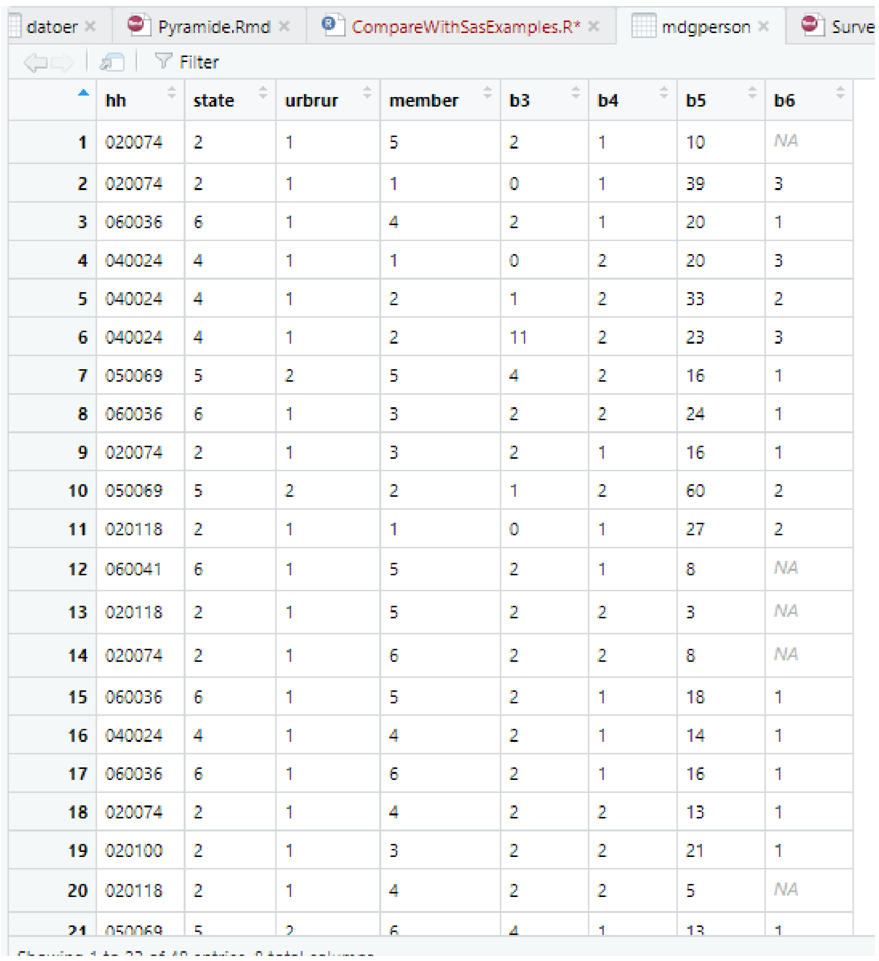
```
'data.frame':    48 obs. of  8 variables:
 $ hh    : chr  "020074" "020074" "060036" "040024" ...
 $ state : chr  "2" "2" "6" "4" ...
 $ urbrur: chr  "1" "1" "1" "1" ...
 $ member: num  5 1 4 1 2 2 5 3 3 2 ...
 $ b3    : num  2 0 2 0 1 11 4 2 2 1 ...
 $ b4    : num  1 1 1 2 2 2 2 2 1 2 ...
 $ b5    : num  10 39 20 20 33 23 16 24 16 60 ...
 $ b6    : num  NA 3 1 3 2 3 1 1 1 2 ...
```

It shows that mdgperson is a data frame with 48 observations (rows) and 8 variables (columns). Then each variable is listed with name, type and the values of the first observations. We see for b6 that missing values are represented with the value NA (not available).

To look at the whole file we can use the *View* command:

```
View(mdgperson)
```

Here is the beginning of the mdgperson data frame:



	hh	state	urbrur	member	b3	b4	b5	b6
1	020074	2	1	5	2	1	10	NA
2	020074	2	1	1	0	1	39	3
3	060036	6	1	4	2	1	20	1
4	040024	4	1	1	0	2	20	3
5	040024	4	1	2	1	2	33	2
6	040024	4	1	2	11	2	23	3
7	050069	5	2	5	4	2	16	1
8	060036	6	1	3	2	2	24	1
9	020074	2	1	3	2	1	16	1
10	050069	5	2	2	1	2	60	2
11	020118	2	1	1	0	1	27	2
12	060041	6	1	5	2	1	8	NA
13	020118	2	1	5	2	2	3	NA
14	020074	2	1	6	2	2	8	NA
15	060036	6	1	5	2	1	18	1
16	040024	4	1	4	2	1	14	1
17	060036	6	1	6	2	1	16	1
18	020074	2	1	4	2	2	13	1
19	020100	2	1	3	2	2	21	1
20	020118	2	1	4	2	2	5	NA
21	050069	5	2	6	4	1	13	1

7.5. Python

We use a *read* method to import external files to Python data frames. To import a delimited file, we can use the *read.csv* method. To import a fixed width file, we can use the *read_fwf* method. But first, we should put the path to the data folder into an object, here called *datapath*. This will make easier to move the examples to another folder. We must specify the width of each column (or the start and end position) and we should name the columns as well:

```
datapath = 'H:/MDG/Data/'
mdgperson = pd.read_fwf(
    datapath + 'mdgperson.txt',
    names=['hh', 'state', 'urbrur', 'member', 'b3', 'b4', 'b5', 'b6'],
    dtype=object,
    na_values={'.', ' .'},
    widths=[6, 1, 1, 1,  2, 1, 2, 1]
)
mdgperson.head(11)
```

To convert missing values in the data file to missing values in Python, NaN (Not a Number) for numbers, we specify na_values. We have also specified that the columns should be objects, which means it can literally contain anything. After the import we list the 11 first rows with the *head* function. Beware that the first row has the index 0:

	hh	state	urbrur	member	b3	b4	b5	b6
0	020074	2	1	5	2	1	10	NaN
1	020074	2	1	1	0	1	39	3
2	060036	6	1	4	2	1	20	1
3	040024	4	1	1	0	2	20	3
4	040024	4	1	2	1	2	33	2
5	040024	4	1	2	11	2	23	3
6	050069	5	2	5	4	2	16	1
7	060036	6	1	3	2	2	24	1
8	020074	2	1	3	2	1	16	1
9	050069	5	2	2	1	2	60	2
10	020118	2	1	1	0	1	27	2

For some information about the data frame, we can use the *info* function:

```
mdgperson.info()
```

The output is like this:

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 48 entries, 0 to 47
Data columns (total 8 columns):
 #   Column  Non-Null Count  Dtype 
---  ------  --------------  ----- 
 0   hh      48 non-null     object
 1   state   48 non-null     object
 2   urbrur  48 non-null     object
 3   member  48 non-null     object
 4   b3      48 non-null     object
 5   b4      48 non-null     object
 6   b5      47 non-null     object
 7   b6      36 non-null     object
dtypes: object(8)
memory usage: 1.6+ KB
```

We see that all columns have been defined as objects, which means they are string columns. When we want to specify the column types ourselves, we can use a dictionary in the *dtype* parameter:

```
mdgperson = pd.read_fwf(
    datapath + 'mdgperson.txt',
    names=['hh', 'state', 'urbrur', 'member', 'b3', 'b4', 'b5', 'b6'],
    dtype={'hh': 'object', 'state': 'object', 'urbrur': 'object', 'b3':
'object', 'b4': 'object', 'b6': 'object'},
    na_values={'.',' .'},
    widths=[6, 1, 1, 1,  2, 1, 2, 1]
)
mdgperson.head(11)
```

Variables not mentioned in the dtype parameter will be given a type based on the content of each column.

When we define a column as int64 the import will fail with an error message if a missing value is found:

```
ValueError: Unable to convert column b5 to type int64
```

Instead, we can import the column as float64 as it accepts missing values.

We only need to specify the columns that will not automatically get the right column types. Columns with letters will by default always be objects, integer values without missing values will be int64 and integers with missing values and decimal numbers will be float64.

8. Getting to know our data

When our file is imported correctly to Sas, Spss, Stata, R or Python, we will use some procedures to get to know our data. We typically use frequency tables and descriptive statistics for this purpose.

8.1. Frequency tables

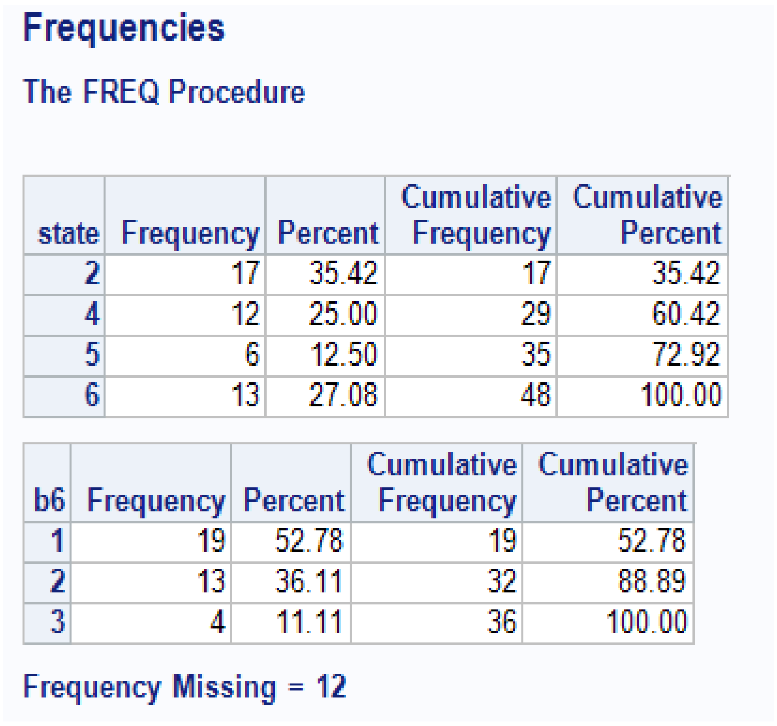
We use frequency tables to get a fast overview of the distribution of variables with few distinct values. In Stata missing values are excluded unless we add the *miss* option. The number of missing values is not listed as they are in Sas and Spss. We may include the missing values as regular values in the tables in Sas by using the *missing* option in the *Tables* statement. In R we use the *exclude=NULL* argument to include NA's in the frequency tables. In Python, we can use the *fillna* method to include missing values in our calculations.

Sas

In base Sas we have to write syntax for this. In Enterprise Guide we find it under Tasks > Describe > One-Way Frequencies and Tasks > Describe > Table Analysis. The written syntax for two One-way frequency and one Two-way frequency tables may look like this (generated syntax from the menus will be more extensive):

```
proc freq data=mdgperson;
 tables state b6 state*b6;
 title "Frequencies";
run;
```

Executed from Enterprise Guide the result will look like this:



Frequencies

The FREQ Procedure

state	Frequency	Percent	Cumulative Frequency	Cumulative Percent
2	17	35.42	17	35.42
4	12	25.00	29	60.42
5	6	12.50	35	72.92
6	13	27.08	48	100.00

b6	Frequency	Percent	Cumulative Frequency	Cumulative Percent
1	19	52.78	19	52.78
2	13	36.11	32	88.89
3	4	11.11	36	100.00

Frequency Missing = 12

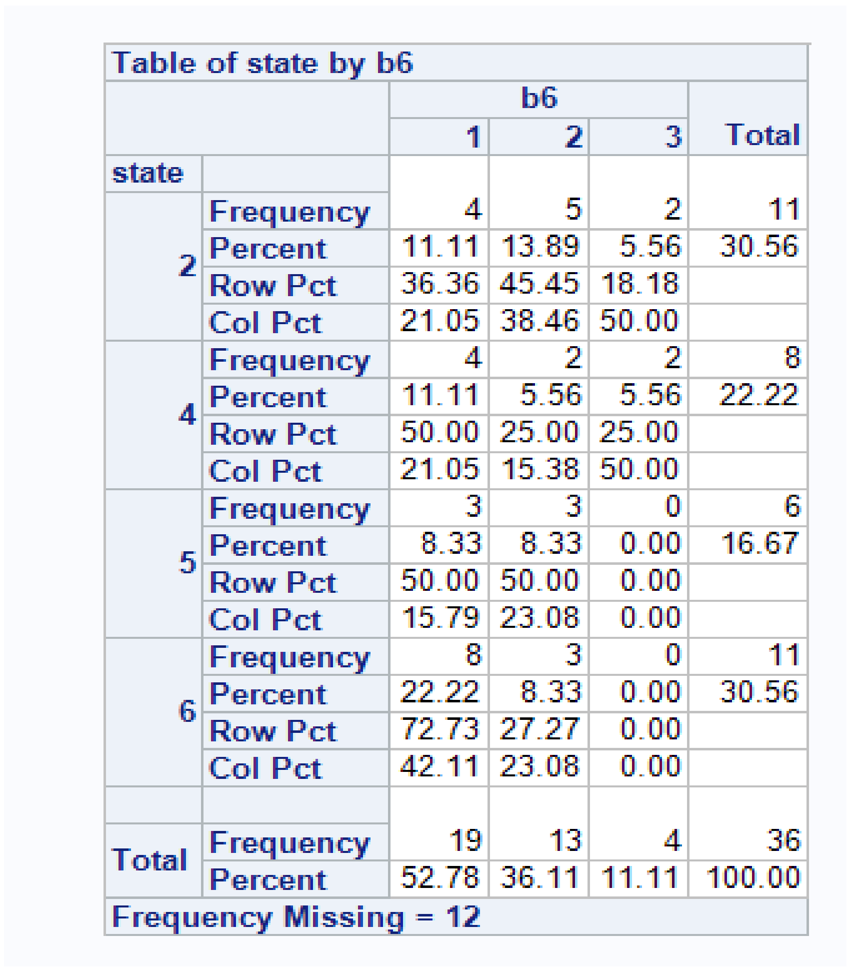


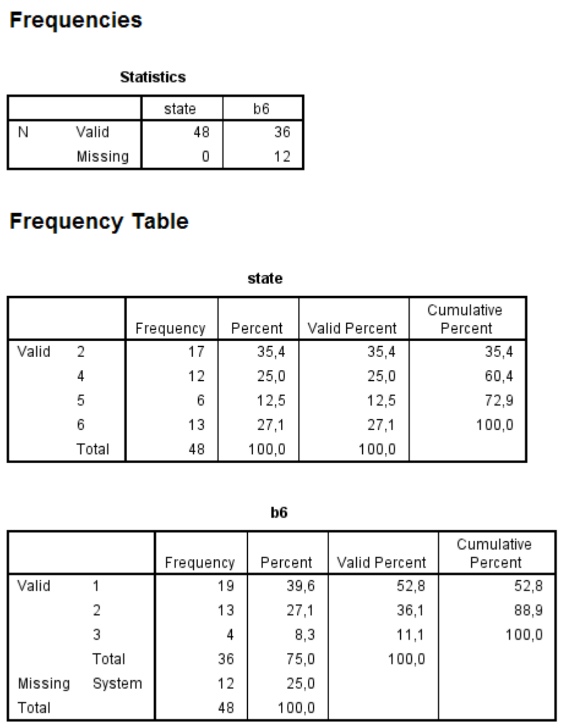
Table of state by b6					
		b6			
		1	2	3	Total
state					
2	Frequency	4	5	2	11
	Percent	11.11	13.89	5.56	30.56
	Row Pct	36.36	45.45	18.18	
	Col Pct	21.05	38.46	50.00	
4	Frequency	4	2	2	8
	Percent	11.11	5.56	5.56	22.22
	Row Pct	50.00	25.00	25.00	
	Col Pct	21.05	15.38	50.00	
5	Frequency	3	3	0	6
	Percent	8.33	8.33	0.00	16.67
	Row Pct	50.00	50.00	0.00	
	Col Pct	15.79	23.08	0.00	
6	Frequency	8	3	0	11
	Percent	22.22	8.33	0.00	30.56
	Row Pct	72.73	27.27	0.00	
	Col Pct	42.11	23.08	0.00	
Total	Frequency	19	13	4	36
	Percent	52.78	36.11	11.11	100.00
Frequency Missing = 12					

Spss

The One-way frequencies in Spss are found in the menu Analyze > Descriptive statistics > Frequencies. For Two-way frequencies we can use Analyze > Descriptive statistics > Crosstabs. When we paste the syntax, it looks like this:

```
FREQUENCIES VARIABLES=state b6
  /ORDER=ANALYSIS.
CROSSTABS
  /TABLES=state BY b6
  /FORMAT=AVALUE TABLES
  /CELLS=COUNT ROW COLUMN TOTAL
  /COUNT ROUND CELL.
```

The frequency tables look like this:



Frequencies

Statistics

		state	b6
N	Valid	48	36
	Missing	0	12

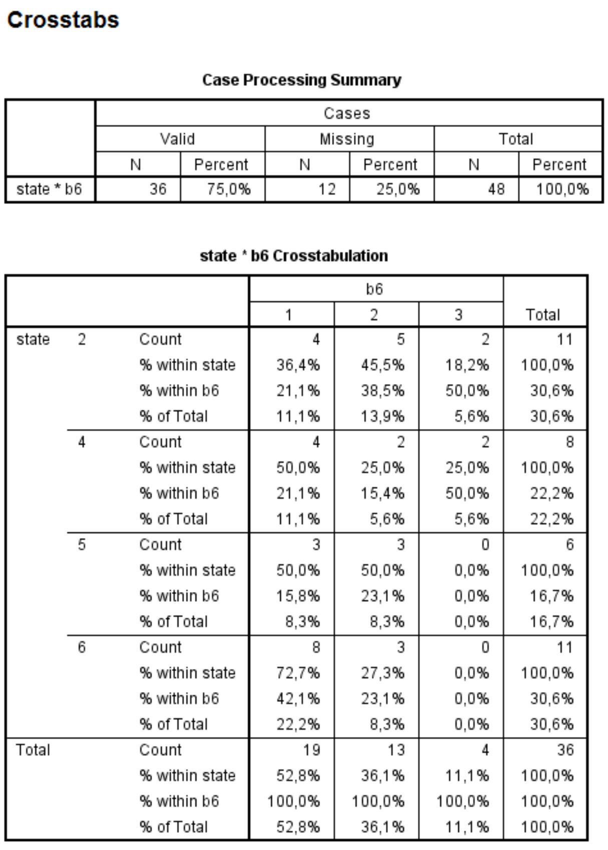
Frequency Table

state

		Frequency	Percent	Valid Percent	Cumulative Percent
Valid	2	17	35,4	35,4	35,4
	4	12	25,0	25,0	60,4
	5	6	12,5	12,5	72,9
	6	13	27,1	27,1	100,0
	Total	48	100,0	100,0	

b6

		Frequency	Percent	Valid Percent	Cumulative Percent
Valid	1	19	39,6	52,8	52,8
	2	13	27,1	36,1	88,9
	3	4	8,3	11,1	100,0
	Total	36	75,0	100,0	
Missing	System	12	25,0		
Total		48	100,0		



Crosstabs

Case Processing Summary

	Cases					
	Valid		Missing		Total	
	N	Percent	N	Percent	N	Percent
state * b6	36	75,0%	12	25,0%	48	100,0%

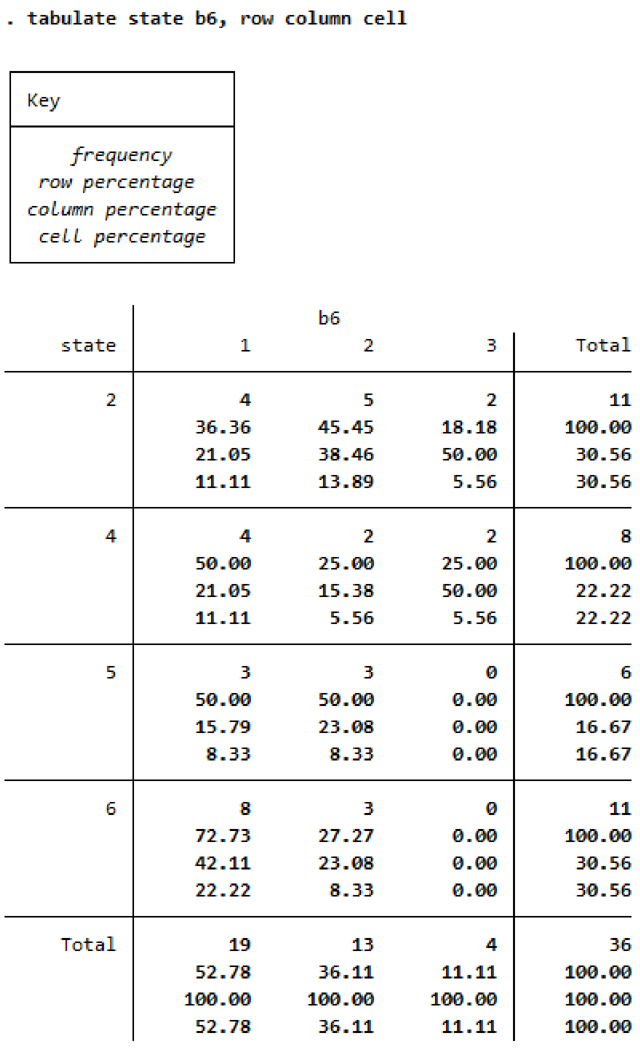
state * b6 Crosstabulation

			b6			Total
			1	2	3	
state	2	Count	4	5	2	11
		% within state	36,4%	45,5%	18,2%	100,0%
		% within b6	21,1%	38,5%	50,0%	30,6%
		% of Total	11,1%	13,9%	5,6%	30,6%
	4	Count	4	2	2	8
		% within state	50,0%	25,0%	25,0%	100,0%
		% within b6	21,1%	15,4%	50,0%	22,2%
		% of Total	11,1%	5,6%	5,6%	22,2%
	5	Count	3	3	0	6
		% within state	50,0%	50,0%	0,0%	100,0%
		% within b6	15,8%	23,1%	0,0%	16,7%
		% of Total	8,3%	8,3%	0,0%	16,7%
	6	Count	8	3	0	11
		% within state	72,7%	27,3%	0,0%	100,0%
		% within b6	42,1%	23,1%	0,0%	30,6%
		% of Total	22,2%	8,3%	0,0%	30,6%
Total		Count	19	13	4	36
		% within state	52,8%	36,1%	11,1%	100,0%
		% within b6	100,0%	100,0%	100,0%	100,0%
		% of Total	52,8%	36,1%	11,1%	100,0%

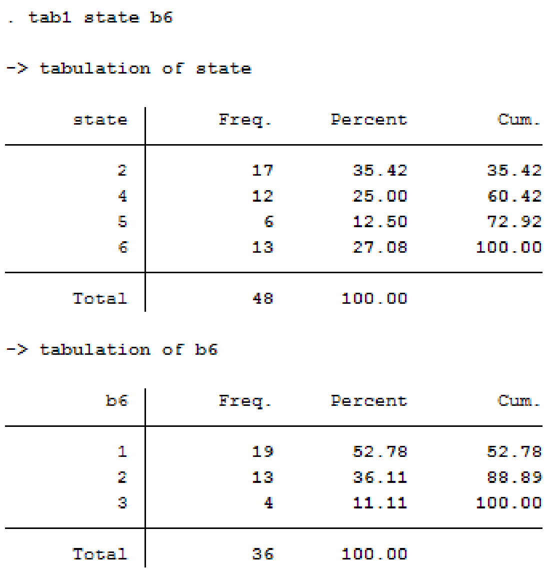
Stata

In Stata we find the one-way frequency here: Statistics > Summaries, tables, and tests > Tables > Multiple one-way tables. The two-way frequencies are here: Statistics > Summaries, tables, and tests > Tables > Two-way tables with measures of association. When we write the commands *tab1* and *tabulate* they will be like this:

```
tab1 state b6
tabulate state b6, row column cell
```



```
. tabulate state b6, row column cell

+-------------------+
| Key               |
|-------------------|
|     frequency     |
|  row percentage   |
| column percentage |
|  cell percentage  |
+-------------------+

           |                 b6
     state |         1          2          3 |     Total
-----------+---------------------------------+----------
         2 |         4          5          2 |        11 
           |     36.36      45.45      18.18 |    100.00 
           |     21.05      38.46      50.00 |     30.56 
           |     11.11      13.89       5.56 |     30.56 
-----------+---------------------------------+----------
         4 |         4          2          2 |         8 
           |     50.00      25.00      25.00 |    100.00 
           |     21.05      15.38      50.00 |     22.22 
           |     11.11       5.56       5.56 |     22.22 
-----------+---------------------------------+----------
         5 |         3          3          0 |         6 
           |     50.00      50.00       0.00 |    100.00 
           |     15.79      23.08       0.00 |     16.67 
           |      8.33       8.33       0.00 |     16.67 
-----------+---------------------------------+----------
         6 |         8          3          0 |        11 
           |     72.73      27.27       0.00 |    100.00 
           |     42.11      23.08       0.00 |     30.56 
           |     22.22       8.33       0.00 |     30.56 
-----------+---------------------------------+----------
     Total |        19         13          4 |        36 
           |     52.78      36.11      11.11 |    100.00 
           |    100.00     100.00     100.00 |    100.00 
           |     52.78      36.11      11.11 |    100.00 
```



```
. tab1 state b6

-> tabulation of state  

      state |      Freq.     Percent        Cum.
------------+-----------------------------------
          2 |         17       35.42       35.42
          4 |         12       25.00       60.42
          5 |          6       12.50       72.92
          6 |         13       27.08      100.00
------------+-----------------------------------
      Total |         48      100.00

-> tabulation of b6  

         b6 |      Freq.     Percent        Cum.
------------+-----------------------------------
          1 |         19       52.78       52.78
          2 |         13       36.11       88.89
          3 |          4       11.11      100.00
------------+-----------------------------------
      Total |         36      100.00
```

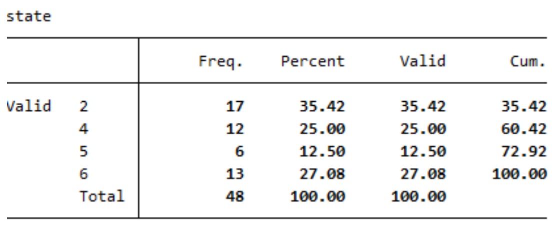
It is also possible to add the *fre* package to Stata. We can do that with this command (when we have Internet connection)

```
ssc install fre
```

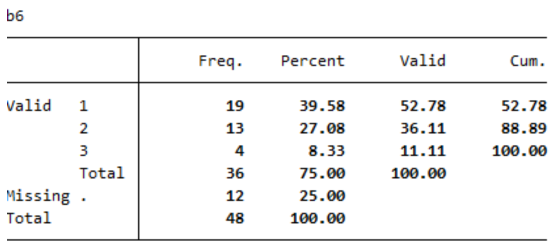
Now we can make one-way frequency tables more like in Sas and Spss:

```
fre state b6
```

The frequency tables:



```
state

               |    Freq.    Percent      Valid       Cum.
---------------+-------------------------------------------
Valid   2      |       17      35.42      35.42      35.42
        4      |       12      25.00      25.00      60.42
        5      |        6      12.50      12.50      72.92
        6      |       13      27.08      27.08     100.00
        Total  |       48     100.00     100.00
---------------+-------------------------------------------
```



```
b6

               |    Freq.    Percent      Valid       Cum.
---------------+-------------------------------------------
Valid   1      |       19      39.58      52.78      52.78
        2      |       13      27.08      36.11      88.89
        3      |        4       8.33      11.11     100.00
        Total  |       36      75.00     100.00
Missing .      |       12      25.00
Total          |       48     100.00
---------------+-------------------------------------------
```

R

There are several different commands for frequency tables in R. One basic command is the *table* command. It shows the distribution of one or more variables. Here is an example on a one-way frequency, beware that the $ sign is used to separate the data frame name from the variable name:

```
table(mdgperson$state)
```

The output is very simple, with no totals (margins). The different values for the variable state in the data frame mdgperson is in the first line of the output and the frequencies are in the second:

```
 2  4  5  6
17 12  6 13
```

To add totals, we use the *addmargins* command. As the R commands are actually functions, we can use a command within another command, like this:

```
addmargins(table(mdgperson$state))
```

We see that we have now added a total to the output listing:

```
  2   4   5   6 Sum
 17  12   6  13  48
```

If we want percentages we can add the *prop.table* command and multiply by 100:

```
addmargins(prop.table (table(mdgperson$state)))*100
```

The percentage table:

```
        2         4         5         6       Sum
 35.41667  25.00000  12.50000  27.08333 100.00000
```

These to outputs may be combined with the *cbind* or *rbind* commands. *cbind* combines columns while *rbind* combine rows:

```
cbind(frequency=addmargins(table(mdgperson$state)),percent=addmargins(prop.ta
ble (table(mdgperson$state)))*100)

rbind(frequency=addmargins(table(mdgperson$state)),percent=addmargins(prop.ta
ble (table(mdgperson$state)))*100)
```

The 2 outputs, the first as columns and the second as rows:

```
    frequency   percent

2          17  35.41667

4          12  25.00000

5           6  12.50000

6          13  27.08333

Sum        48 100.00000

                 2  4    5        6 Sum

frequency 17.00000 12  6.0 13.00000  48

percent   35.41667 25 12.5 27.08333 100
```

The default for these frequency tables is to omit the NA values. If we want the NA's, we add the *exclude* argument:

```
table(mdgperson$b6,exclude = NULL)
```

We see that we have a column fro NA:

```
   1    2    3 <NA>

  19   13    4   12
```

To add cumulative frequencies, we can use the *transform* command which add them after the frequency table is converted to a data frame. To convert the output table to a data frame we use the *as.data.frame* function. The data frame will have to variables, *state* and *Freq*. We calculate the cumulative frequencies by using the *cumsum* function for the *Freq* variable. The syntax looks like this:

```
transform(as.data.frame(table(state=mdgperson$state)),cum_freq=cumsum(Freq))
```

This gives us an output table:

```
  state Freq cum_freq

1     2   17       17

2     4   12       29

3     5    6       35

4     6   13       48
```

Now we can make a table with both cumulative frequencies and percentages and missing values included:

```
transform(as.data.frame(table(b6=mdgperson$b6,exclude=NULL)),percentage=Freq/
nrow(mdgperson)*100,cum_freq=cumsum(Freq),cum_pct=cumsum(Freq/nrow(mdgperson)
*100))
```

We use the *nrow* function to find the number of rows in the data frame. This is the output:

```
    b6 Freq percentage cum_freq   cum_pct
1    1   19  39.583333       19  39.58333
2    2   13  27.083333       32  66.66667
3    3    4   8.333333       36  75.00000
4 <NA>   12  25.000000       48 100.00000
```

We can use the table command for two-way frequencies also. In this example we include totals and NA values:

```
addmargins(table(mdgperson$state,mdgperson$b6,exclude=NULL))
```

The two-way frequency with totals and NA's:

```
       1  2  3 <NA> Sum
  2    4  5  2    6  17
  4    4  2  2    4  12
  5    3  3  0    0   6
  6    8  3  0    2  13
  Sum 19 13  4   12  48
```

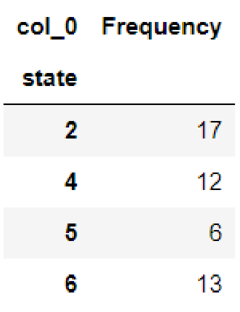
There are other commands for frequency tables in other R packages, like *summarytools* and *tabular*.

Python

We can use the *crosstab* function to create simple frequency tables:

```
pd.crosstab(mdgperson.state, columns="Frequency")
```

This will give us a table like this:



col_0	Frequency
state	
2	17
4	12
5	6
6	13

We can add totals to the table:

```
pd.crosstab(mdgperson.state, columns='Frequency', margins=True)
```

There will be totals both in the columns and rows:

```
col_0  Frequency  All
state
    2         17   17
    4         12   12
    5          6    6
    6         13   13
  All         48   48
```

When we want percentages and cumulative counts and percentages we can calculate them, for instance like this:

```
freqvar = 'b6'
freq = mdgperson[freqvar].value_counts(dropna=False).sort_index()
freq = pd.DataFrame({
    freqvar: freq.index,
    'Frequency': freq.values,
    'Percent': ((freq.values/freq.values.sum())*100).round(2),
    'Cumulative Frequency': freq.values.cumsum(),
    'Cumulative Percent':
((freq.values.cumsum()/freq.values.sum())*100).round(2)
}
)
freq
```

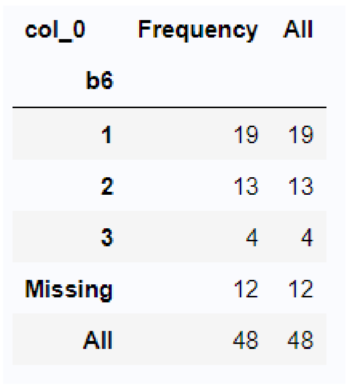
The output will be like this:

```
    b6  Frequency  Percent  Cumulative Frequency  Cumulative Percent
0    1         19    39.58                    19               39.58
1    2         13    27.08                    32               66.67
2    3          4     8.33                    36               75.00
3  NaN         12    25.00                    48              100.00
```

If there are NaN values in the categories they will not be included as separate rows or columns unless we change the NaN to a valid value, for instance the text Missing. Here is an example:

```
pd.crosstab(mdgperson['b6'].fillna('Missing'), columns='Frequency',
margins=True)
```

The Nan values in the categories are now included as a Missing category as we have filled them with the text Missing:



col_0	Frequency	All
b6		
1	19	19
2	13	13
3	4	4
Missing	12	12
All	48	48

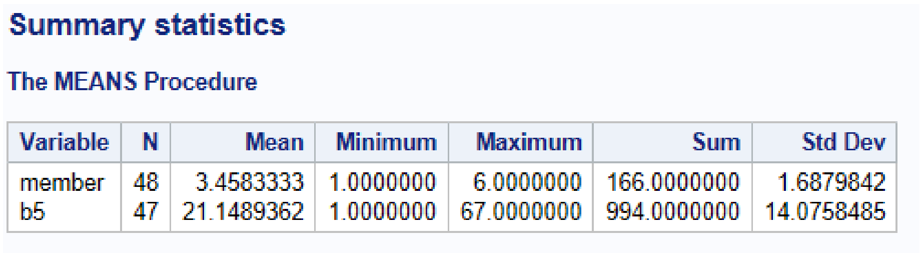
8.2. Descriptive statistics

These are the naming of basic statistics made on numeric variables:

Statistic	Sas	Spss	Stata	R	Python
Maximum	Max	max	max	max	max
Minimum	Min	min	min	min	min
Average	Mean	mean	mean	mean	mean
Sum	Sum	sum	sum	sum	sum
Number of valid values	N	n	n or count	count	count
Number of missing values	Nmiss	N/A	N/A	~sum(is.na(.)) (when using package dplyr)	N/A
Standard deviation	Std	stddev	sd	sd	std

Sas

The basic procedure used in Sas for descriptive statistics is *Proc means*. It is found in the menu Tasks > Describe > Summary statistics. The output for 2 variables with some different statistics is like this:



Summary statistics

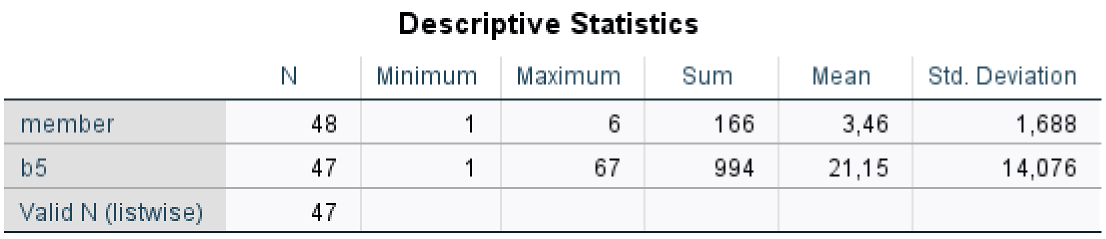
The MEANS Procedure

Variable	N	Mean	Minimum	Maximum	Sum	Std Dev
member	48	3.4583333	1.0000000	6.0000000	166.0000000	1.6879842
b5	47	21.1489362	1.0000000	67.0000000	994.0000000	14.0758485

When we write syntax, it will look like this:

```
proc means data=mdgperson n mean min max sum std;
 var member b5;
 title "Summary statistics";
run;
```

Spss
In Spss we use the *Descriptives* command, and it is found under Analyze > Descriptive statistics > Descriptives. The same statistics as above will be like this in Spss:



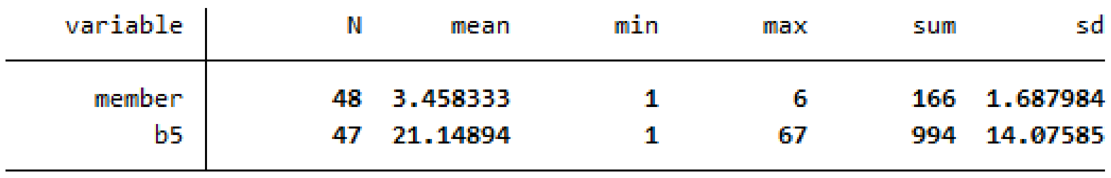
Descriptive Statistics

	N	Minimum	Maximum	Sum	Mean	Std. Deviation
member	48	1	6	166	3,46	1,688
b5	47	1	67	994	21,15	14,076
Valid N (listwise)	47					

The Spss syntax:

```
DESCRIPTIVES VARIABLES=member b5
  /STATISTICS= MEAN MIN MAX SUM STDDEV.
```

Stata
Stata has descriptive statistics under Statistics > Summaries, tables, and tests > Tables > Table of summary statistics (tabstat):



```
    variable |         N      mean       min       max       sum        sd
-------------+------------------------------------------------------------
      member |        48  3.458333         1         6       166  1.687984
          b5 |        47  21.14894         1        67       994  14.07585
--------------------------------------------------------------------------
```

We can write the code instead of using the menus:

tabstat member b5, statistics(count mean min max sum sd) columns(statistics)

R
To get basic descriptives on a data frame we can use the summary command:

```
summary(mdgperson)
```

The output will be like this:

```
      hh                 state                urbrur               member     
 Length:48          Length:48          Length:48          Min.   :1.000  
 Class :character   Class :character   Class :character   1st Qu.:2.000  
 Mode  :character   Mode  :character   Mode  :character   Median :3.500  
                                                           Mean   :3.458  
                                                           3rd Qu.:5.000  
                                                           Max.   :6.000  
```

```
       b3               b4                b5                b6
 Min.   : 0.000   Min.   :1.000   Min.   : 1.00   Min.   :1.000
 1st Qu.: 1.000   1st Qu.:1.000   1st Qu.:11.50   1st Qu.:1.000
 Median : 2.000   Median :2.000   Median :18.00   Median :1.000
 Mean   : 2.667   Mean   :1.562   Mean   :21.15   Mean   :1.583
 3rd Qu.: 2.000   3rd Qu.:2.000   3rd Qu.:25.50   3rd Qu.:2.000
 Max.   :11.000   Max.   :2.000   Max.   :67.00   Max.   :3.000
                                  NA's   :1       NA's   :12
```

For character variables we only get information about the number of rows (Length: 48). For numeric variables we get minimum, 1st quantile, median, 3rd quantile, maximum and NA's.

For more specific descriptives we may use the *dplyr* package. If it is not installed it can be installed with the *install.packages* command

```
install.packages("dplyr")
```

Packages that are not included in the basic R language need to be loaded and attached by using the *library* command:

```
library(dplyr)
```

Now we can use it to for instance list some descriptive statistics. The *dplyr* package can pipe commands together and we use %>% as the pipe. First, we choose our input data frame. Then we pipe it to our descriptives command:

```
mdgperson %>%
   summarize_if(is.numeric,c("sum","mean","sd","min","max"))
```

We have chosen that we only want to use the numeric variables in the data frame by using the is.numeric function, which will select only the numeric variables. The result is like this:

```
  member_sum b3_sum b4_sum b5_sum b6_sum member_mean  b3_mean b4_mean b5_mean
1        166    128     75     NA     NA    3.458333 2.666667  1.5625      NA
  b6_mean member_sd    b3_sd    b4_sd b5_sd b6_sd member_min b3_min b4_min
1      NA  1.687984 3.068948 0.501328    NA    NA          1      0      1
  b5_min b6_min member_max b3_max b4_max b5_max b6_max
1     NA     NA          6     11      2     NA     NA
```

Variables with NA's will have NA for all the descriptives. To avoid the NA's we can omit with the argument *na.rm=TRUE*:

```
mdgperson %>%
   summarize_if(is.numeric,c("sum","mean","sd","min","max"),na.rm=TRUE)
```

Now we get figures for all the variables:

```
  member_sum b3_sum b4_sum b5_sum b6_sum member_mean  b3_mean b4_mean  b5_mea
n

1        166    128     75    994     57    3.458333 2.666667  1.5625 21.1489
4

   b6_mean member_sd    b3_sd    b4_sd    b5_sd     b6_sd member_min b3_min

1 1.583333  1.687984 3.068948 0.501328 14.07585 0.6917886          1      0

  b4_min b5_min b6_min member_max b3_max b4_max b5_max b6_max

1      1      1      1          6     11      2     67      3
```

If we want to count the NA's we can do like this:

```
mdgperson %>%
   summarise_if(is.numeric,~sum(is.na(.)))
```

The *is.na* function returns TRUE when a value is NA and FALSE when it is not. The number of true values will then be added up for each numeric variable (the dot within *is.na* symbolize all variables within a data frame and *is.numeric* limit it to the numeric variables. The result is here:

```
  member b3 b4 b5 b6

1      0  0  0  1 12
```

We see that the names of the counts of NA's are the same as the original variable names. Now we can combine the descriptives into one data frame. To do that we first put each of the descriptives in separate data frames. Then we rename the names of the NA counts with the *colnames* and *paste0* commands. Finally, we combine them together with the *cbind* command:

```
desc<-mdgperson %>%
   summarize_if(is.numeric,c("sum","mean","sd","min","max"),na.rm=TRUE)
nnas<-mdgperson %>%
   summarise_if(is.numeric,~sum(is.na(.)))
colnames(nnas) <- paste0(colnames(nnas),'_na')
cbind(desc,nnas)
```

First, we create two data frames, *desc* and *nnas*. Then we change the column names for the nnas data frame so that we add the text _na at the end of each column. We do that with the *paste0* function which concatenate texts together (the 0 in *paste0* tells us that there shall be no spaces between the concatenated texts). When there is more than one element in the first argument (*colnames(nnas)*), the second argument will be added to all from the first argument. Hence all the column names will be changed. Finally, we combine the columns together with the *cbind* command:

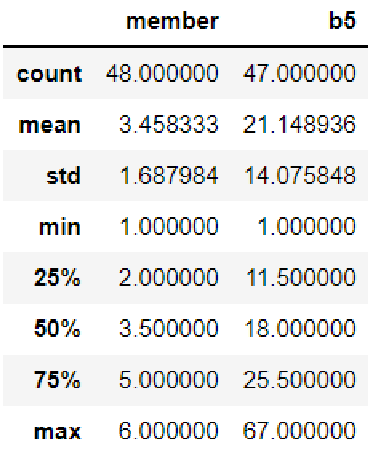
```
  member_sum b3_sum b4_sum b5_sum b6_sum member_mean  b3_mean b4_mean  b5_mean

1        166    128     75    994     57    3.458333 2.666667  1.5625 21.14894

   b6_mean member_sd    b3_sd    b4_sd    b5_sd     b6_sd member_min b3_min

1 1.583333  1.687984 3.068948 0.501328 14.07585 0.6917886          1      0

  b4_min b5_min b6_min member_max b3_max b4_max b5_max b6_max member_na b3_na

1      1      1      1          6     11      2     67      3         0     0

  b4_na b5_na b6_na

1     0     1    12
```

Python

We can use the *describe* method in Pandas to get a fast and brief descriptive overview of our numeric columns. The syntax is as simple as this:

```
mdgperson.describe()
```

The result is shown here:

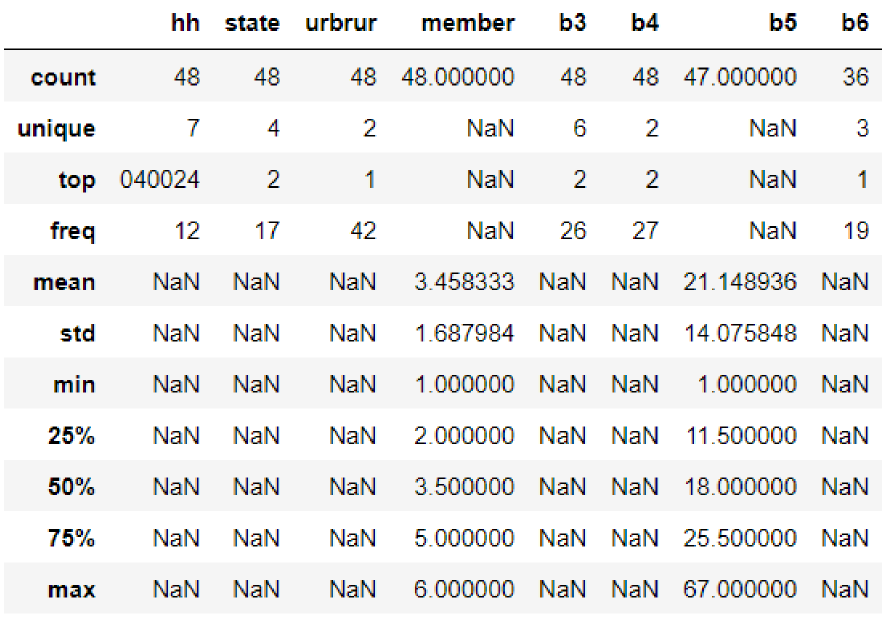


	member	b5
count	48.000000	47.000000
mean	3.458333	21.148936
std	1.687984	14.075848
min	1.000000	1.000000
25%	2.000000	11.500000
50%	3.500000	18.000000
75%	5.000000	25.500000
max	6.000000	67.000000

If we want all columns, we can do like this:

```
mdgperson.describe(include='all')
```

There will be some other statistics for the string columns in the output:



	hh	state	urbrur	member	b3	b4	b5	b6
count	48	48	48	48.000000	48	48	47.000000	36
unique	7	4	2	NaN	6	2	NaN	3
top	040024	2	1	NaN	2	2	NaN	1
freq	12	17	42	NaN	26	27	NaN	19
mean	NaN	NaN	NaN	3.458333	NaN	NaN	21.148936	NaN
std	NaN	NaN	NaN	1.687984	NaN	NaN	14.075848	NaN
min	NaN	NaN	NaN	1.000000	NaN	NaN	1.000000	NaN
25%	NaN	NaN	NaN	2.000000	NaN	NaN	11.500000	NaN
50%	NaN	NaN	NaN	3.500000	NaN	NaN	18.000000	NaN
75%	NaN	NaN	NaN	5.000000	NaN	NaN	25.500000	NaN
max	NaN	NaN	NaN	6.000000	NaN	NaN	67.000000	NaN

The *top* category shows the value with highest frequency and the *freq* category contains the number of rows the top value has.

8.3. Descriptive statistics grouped

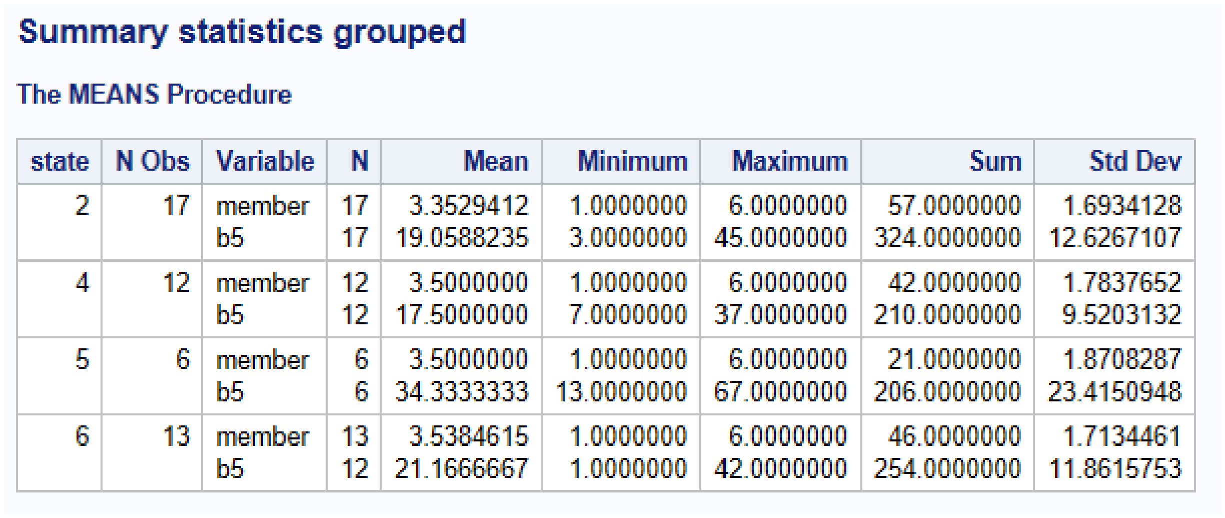
We can also group the descriptive statistics.

Sas

In Sas we add the *Class* statement to the *Proc means* procedure like this:

```
proc means data=mdgperson n mean min max sum std;
 class state;
 var member b5;
 title "Summary statistics grouped";
run;
```

The program gives this output:



Summary statistics grouped

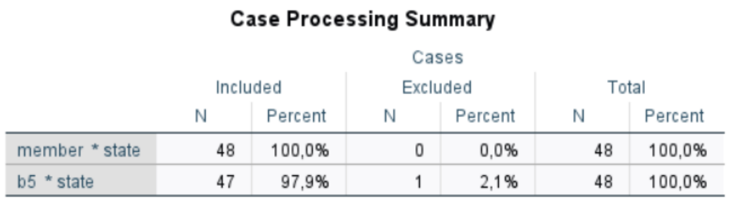
The MEANS Procedure

state	N Obs	Variable	N	Mean	Minimum	Maximum	Sum	Std Dev
2	17	member b5	17 17	3.3529412 19.0588235	1.0000000 3.0000000	6.0000000 45.0000000	57.0000000 324.0000000	1.6934128 12.6267107
4	12	member b5	12 12	3.5000000 17.5000000	1.0000000 7.0000000	6.0000000 37.0000000	42.0000000 210.0000000	1.7837652 9.5203132
5	6	member b5	6 6	3.5000000 34.3333333	1.0000000 13.0000000	6.0000000 67.0000000	21.0000000 206.0000000	1.8708287 23.4150948
6	13	member b5	13 12	3.5384615 21.1666667	1.0000000 1.0000000	6.0000000 42.0000000	46.0000000 254.0000000	1.7134461 11.8615753

Spss

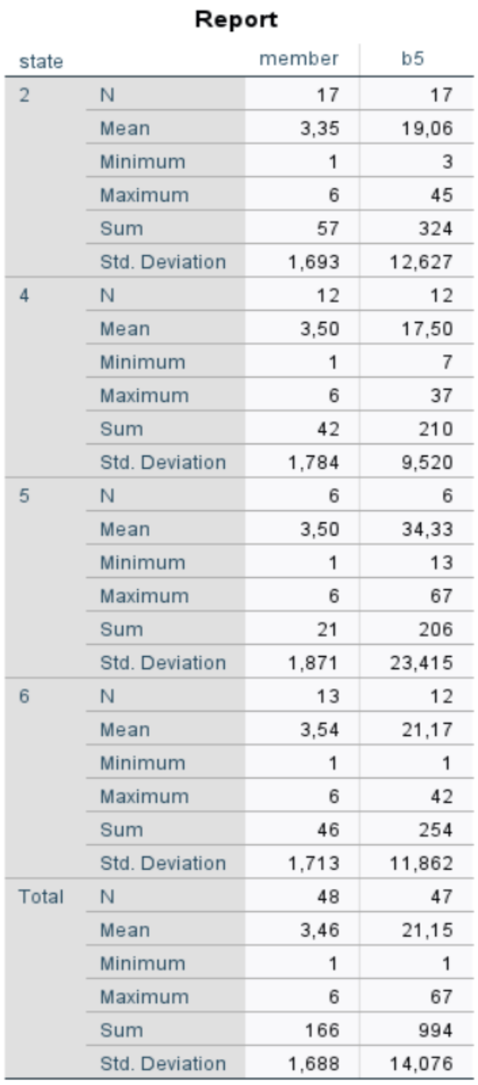
To group the descriptive statistics in Spss we switch to the *Means* procedure (Analyze > Compare means > Means):

```
MEANS TABLES=member b5 BY state
  /CELLS COUNT MEAN MIN MAX SUM STDDEV.
```



Case Processing Summary

	Cases					
	Included		Excluded		Total	
	N	Percent	N	Percent	N	Percent
member * state	48	100,0%	0	0,0%	48	100,0%
b5 * state	47	97,9%	1	2,1%	48	100,0%



Report

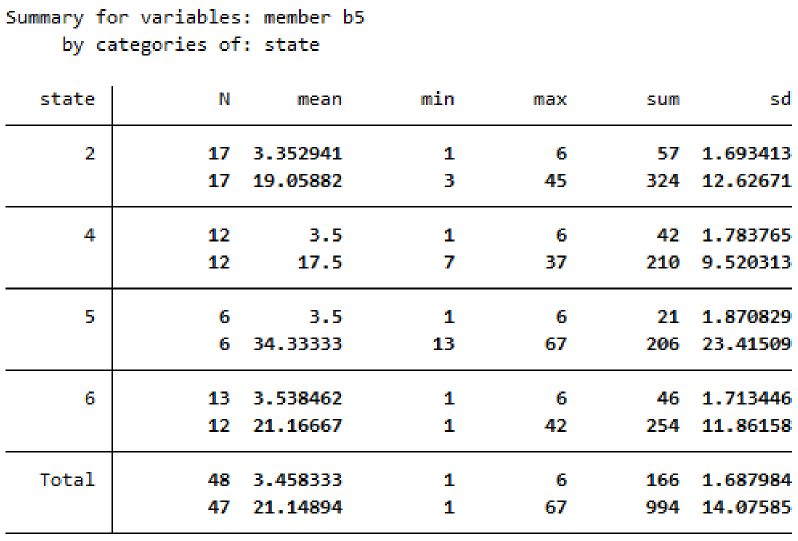
state		member	b5
2	N	17	17
	Mean	3,35	19,06
	Minimum	1	3
	Maximum	6	45
	Sum	57	324
	Std. Deviation	1,693	12,627
4	N	12	12
	Mean	3,50	17,50
	Minimum	1	7
	Maximum	6	37
	Sum	42	210
	Std. Deviation	1,784	9,520
5	N	6	6
	Mean	3,50	34,33
	Minimum	1	13
	Maximum	6	67
	Sum	21	206
	Std. Deviation	1,871	23,415
6	N	13	12
	Mean	3,54	21,17
	Minimum	1	1
	Maximum	6	42
	Sum	46	254
	Std. Deviation	1,713	11,862
Total	N	48	47
	Mean	3,46	21,15
	Minimum	1	1
	Maximum	6	67
	Sum	166	994
	Std. Deviation	1,688	14,076

Stata

In Stata we stick to the *tabstat* command and add the By-group:

```
tabstat member b5, by (state) statistics(count mean min max sum sd)
columns(statistics)
```

Here is the table:


```
Summary for variables: member b5
     by categories of: state

   state |        N      mean       min       max       sum        sd
---------+------------------------------------------------------------
       2 |       17  3.352941         1         6        57  1.693413
         |       17  19.05882         3        45       324  12.62671
---------+------------------------------------------------------------
       4 |       12       3.5         1         6        42  1.783765
         |       12      17.5         7        37       210  9.520313
---------+------------------------------------------------------------
       5 |        6       3.5         1         6        21  1.870829
         |        6  34.33333        13        67       206  23.41509
---------+------------------------------------------------------------
       6 |       13  3.538462         1         6        46  1.713446
         |       12  21.16667         1        42       254  11.86158
---------+------------------------------------------------------------
   Total |       48  3.458333         1         6       166  1.687984
         |       47  21.14894         1        67       994  14.07585
---------+------------------------------------------------------------
```

1.1.1 R

We continue with the *dplyr* package when we want grouped statistics. Now we use *summarize_at* with the *vars* argument to choose which variables we want to use:

```
mdgperson %>%
   group_by(state) %>%
   summarize_at(vars(member,b5),c("sum","mean","sd","min","max"),na.rm=TRUE)
```

We group by state and calculate sum, mean, standard deviation, minimum and maximum values for member and b6 and remove NA values. The result will be a data frame which looks like this:

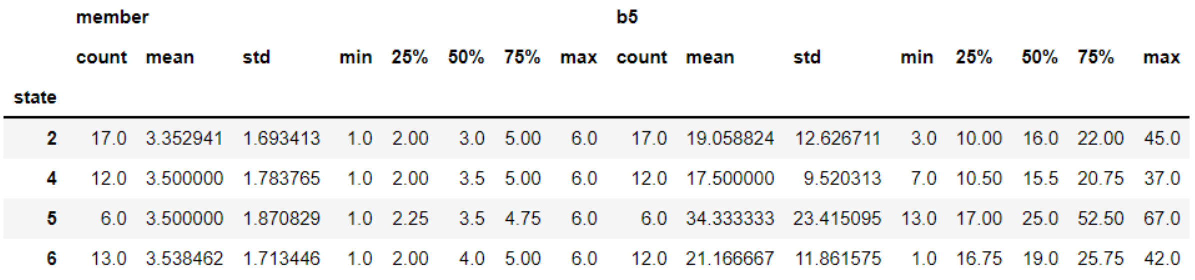
```
# A tibble: 4 x 11

  state member_sum b5_sum member_mean b5_mean member_sd b5_sd member_min b5_min member_max b5_max

  <chr>      <dbl>  <dbl>       <dbl>   <dbl>     <dbl> <dbl>      <dbl>  <dbl>      <dbl>  <dbl>

1 2             57    324        3.35    19.1      1.69 12.6           1      3          6     45
2 4             42    210        3.5     17.5      1.78  9.52          1      7          6     37
3 5             21    206        3.5     34.3      1.87 23.4           1     13          6     67
4 6             46    254        3.54    21.2      1.71 11.9           1      1          6     42
```

Python

We can also have our descriptive statistics grouped. Here is an example where we group by state:

```
mdgperson.groupby(mdgperson['state']).describe()
```

The result is now by state, and the statistics are transposed:

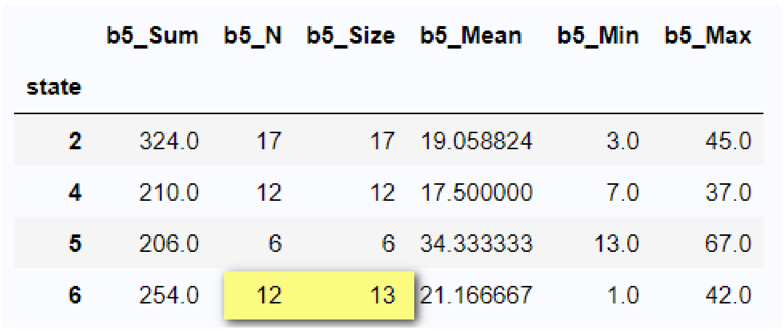


	member								b5							
	count	mean	std	min	25%	50%	75%	max	count	mean	std	min	25%	50%	75%	max
state																
2	17.0	3.352941	1.693413	1.0	2.00	3.0	5.00	6.0	17.0	19.058824	12.626711	3.0	10.00	16.0	22.00	45.0
4	12.0	3.500000	1.783765	1.0	2.00	3.5	5.00	6.0	12.0	17.500000	9.520313	7.0	10.50	15.5	20.75	37.0
5	6.0	3.500000	1.870829	1.0	2.25	3.5	4.75	6.0	6.0	34.333333	23.415095	13.0	17.00	25.0	52.50	67.0
6	13.0	3.538462	1.713446	1.0	2.00	4.0	5.00	6.0	12.0	21.166667	11.861575	1.0	16.75	19.0	25.75	42.0

The sum of the values is not included in this overview. To calculate the sum, we can use the *agg* function:

```
mdgperson.groupby(mdgperson['state']).agg(b5_Sum=('b5', 'sum'),
                                          b5_N=('b5', 'count'),
                                          b5_Size=('b5', 'size'),
                                          b5_Mean=('b5', 'mean'),
                                          b5_Min=('b5', 'min'),
                                          b5_Max=('b5', 'max')
                                         )
```

Beware the difference between *count* and *size*. *count* omits NaN's in the calculation while *size* includes them:

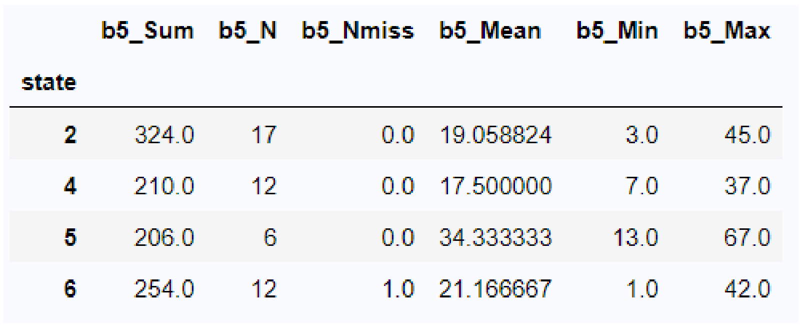


state	b5_Sum	b5_N	b5_Size	b5_Mean	b5_Min	b5_Max
2	324.0	17	17	19.058824	3.0	45.0
4	210.0	12	12	17.500000	7.0	37.0
5	206.0	6	6	34.333333	13.0	67.0
6	254.0	12	13	21.166667	1.0	42.0

If we want to calculate the number of missing values, there are no built-in aggregation function for this. However, we can create our own functions within the aggregation. They are called lambda functions. To calculate the number of missing values we subtract the *count* from the *size.* Beware that there should be no brackets after size, however after count we need brackets:

```
mdgperson.groupby(mdgperson['state']).agg(b5_Sum=('b5', 'sum'),
                                          b5_N=('b5', 'count'),
                                          b5_Nmiss=('b5', lambda x: (x).size-
(x).count()),

                                          b5_Mean=('b5', 'mean'),
                                          b5_Min=('b5', 'min'),
                                          b5_Max=('b5', 'max')
                                          )
```

The number of missing values is now listed as a separate column:



state	b5_Sum	b5_N	b5_Nmiss	b5_Mean	b5_Min	b5_Max
2	324.0	17	0.0	19.058824	3.0	45.0
4	210.0	12	0.0	17.500000	7.0	37.0
5	206.0	6	0.0	34.333333	13.0	67.0
6	254.0	12	1.0	21.166667	1.0	42.0

9. Conditions

Conditions are widely used in programming. We often want to do actions based on conditions. These actions are for instance to assign values to new variables or to select which data to read or write. A condition will be based on expressions. An expression will usually consist of one or more operators, as described on page 18, and one or more variables and sometimes functions. We will now look at how we use conditions when we read and write data.

9.1. Sas

An *If* statement is only allowed in the Data step and is used for a condition after an observation is read. If we want to select observations upon reading, we use a *Where* statement. The *Where* statement may be used in the Data step and most of the procedures. This means that we don't have to make a subset of the data before we make a frequency table. We may simply use a *Where* statement with the selection condition in the procedure:

```
proc freq data=mdgperson;
 where b5 >= 11;
 table b4 /missing;
 title 'Persons aged 11 years and above';
run;
```

If we want to make a new dataset as a subset of the data, we can use a *Data* step like this:

```
data heads;
 set mdgperson;
 where b3 = 0;
run;
```

The *Where* statement selects the observations when the data is read. This is more efficient compared to using an *If* statement instead because the observation which is finally selected is kept during all Data step statements. This program gives the same result as the one above, but it will use more time:

```
data heads;
 set mdgperson;
 if b3 = 0 then
  output;
run;
```

If statements for selection should only be used when we use variables made in the actual *Data* step in our conditions.

9.2. Spss

The command for selecting data in Spss is *Select if*. It will select which observations to keep. We can make a temporary subset of the data if we use it together with the *Temporary* command. The whole dataset is available again after the *Frequency* command. For a frequency table based on a subset of data we can do like this:

```
TEMPORARY.
SELECT IF (b5 >= 11).
FREQUENCIES b4.
```

If we want a permanent subset of our data, we drop the *Temporary* command and add a *Save* command instead:

```
SELECT IF (b3 = 0).
SAVE OUTFILE='h:\mdg\data\heads.sav'.
```

9.3. Stata

For the frequency table we can make a subset by adding an *if* qualifier:

```
tab1 b4 if b5 >= 11
```

However, this qualifier includes the observations with missing values. To avoid this, we can add one more condition like this:

```
tab1 b4 if b5 >= 11 & b5 <115
```

To create a permanent subset of our data, we can use the *keep* and *save* commands:

```
keep if b3 == 0
```

```
save "h:\mdg\data\heads.dta", replace
```

9.4. R

We can use the *subset* command in R when we create our tables. Here we tell that we will make a frequency table for the variable state in the mdgperson data frame. We also use the *subset* command to select rows where the variable b5 is greater than 11. NA's are excluded:

```
table(state=subset(mdgperson$state,mdgperson$b5 > 11),exclude=NA)
```

To create a new data frame with a subset of the data we can also use subset:

```
heads <- subset(mdgperson,b3 == 0)
```

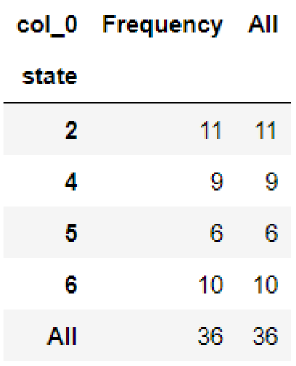
For more use of conditions, see the chapter about missing values, page 139.

9.5. Python

We can use the *loc* method to select the rows we want to use in our table. It is connected to the column we will distribute the table by:

```
pd.crosstab(mdgperson.loc[mdgperson['b5'] >= 11, 'state'],
columns='Frequency', margins=True)
```

Now only those where *b5* >= 11 are included in the table:



col_0	Frequency	All
state		
2	11	11
4	9	9
5	6	6
6	10	10
All	36	36

When we have a crosstab with two columns, we only need the selection once:

```
pd.crosstab(mdgperson.loc[mdgperson['b5'] >= 11, 'state'],
mdgperson['urbrur'], margins=True)
```

We can create a new data frame where we select rows according to a condition:

```
heads = mdgperson.loc[mdgperson['b3'] == '0']
heads
```

When we have more than one condition, we should use parenthesises, otherwise we may get an error message like this:

```
TypeError: Cannot perform 'rand_' with a dtyped [float64] array and scalar
of type [bool]
```

This is due to the precedence of & and >, where & has the higher. Here is an example where we want list heads less than 30 years old:

```
mdgperson.loc[(mdgperson['b3'] == '0') & (mdgperson['b5'] < 30)]
```

When we want to select based on a list of values for a column, we can use the *isin* function:

```
mdgperson.loc[mdgperson['hh'].isin(['040024', '020074'])]
```

Instead of using the loc method we can use the query method. The *loc* selections above will be queries like this:

```
heads = mdgperson.query('b3 == "0"')
mdgperson.query('b3 == "0" & b5 < 30')
mdgperson.query('hh in ["040024", "020074"]')
```

For the last one we use the in operator instead of the *isin* function that was used above.

10. Dealing with duplicates

There are usually some variables (or just one) that will identify each observation in our datasets. These variables should have unique values, if not the dataset contains duplicates. Duplicates cause problems when we match datasets and in other data processing and should be avoided. There are several reasons for duplicates in a dataset:

- Double data entry
- The same identification used for two different units
- Mistyping of the identification
- Programming errors

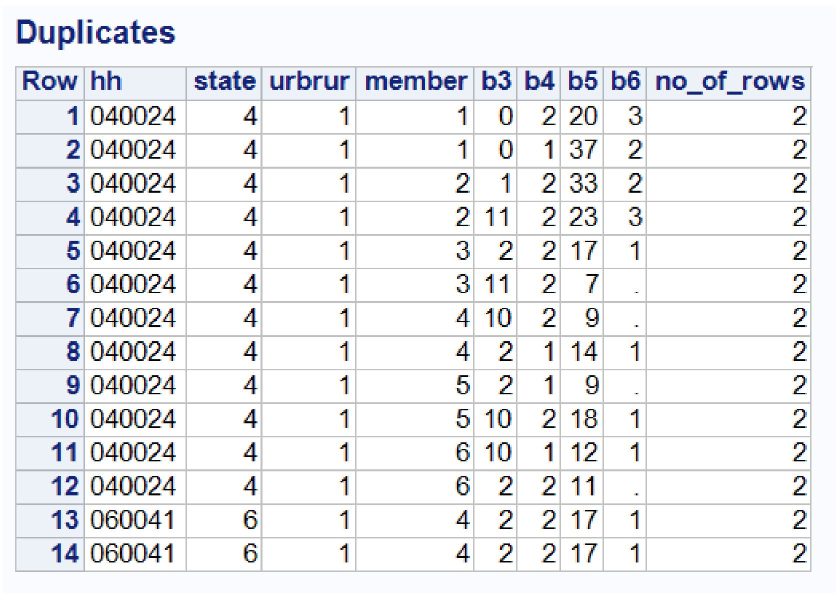
To get rid of duplicates we can use some programming syntax. First, we have to find which duplicates we have. Then we can delete the true duplicates (where all corresponding variables have the same value for more than one observation). After we have deleted true duplicates, we find out how to deal with the remaining duplicates and write syntax for doing these corrections.

10.1. Sas

We need a few lines of code to list the duplicates in Sas:

```
proc sql number;
 title 'Duplicates';
 select *, count(1) as no_of_rows
 from mdgperson
 group by hh, member
 having no_of_rows > 1
 order by hh, member
 ;
quit;
```

This list will be produced:



Duplicates

Row	hh	state	urbrur	member	b3	b4	b5	b6	no_of_rows
1	040024	4	1	1	0	2	20	3	2
2	040024	4	1	1	0	1	37	2	2
3	040024	4	1	2	1	2	33	2	2
4	040024	4	1	2	11	2	23	3	2
5	040024	4	1	3	2	2	17	1	2
6	040024	4	1	3	11	2	7	.	2
7	040024	4	1	4	10	2	9	.	2
8	040024	4	1	4	2	1	14	1	2
9	040024	4	1	5	2	1	9	.	2
10	040024	4	1	5	10	2	18	1	2
11	040024	4	1	6	10	1	12	1	2
12	040024	4	1	6	2	2	11	.	2
13	060041	6	1	4	2	2	17	1	2
14	060041	6	1	4	2	2	17	1	2

By looking at this list we find that for household 040024 there are duplicates for each person. However, the data for each person differ between the duplicates. It looks like one of the households has got a wrong household number. This should be checked and corrected. For household number

060041 we see that all variables have the same data for both observations. This looks like doubly entered data, called a true duplicate. This should also be checked and one of them is likely to be deleted. To delete true duplicates, we can use the *noduprecs* option in *Proc sort*. It will delete duplicate records (the output dataset without duplicates is the one called *mdgperson_noduprec*):

```
proc sort data=mdgperson out=mdgperson_noduprec noduprecs;
 by hh member;
run;
```

The log will tell us how many records have been deleted:

```
NOTE: 1 duplicate observations were deleted.
```

If we, after some research, find that one of the household identifications is a mistype of 040034 we can change the identification for those and sort the data again:

```
data mdgperson_nodup;
 set mdgperson_noduprec;
 by hh member;
 if hh = '040024' and first.member then
  do;
   hh = '040034';
  end;
run;
proc sort data=mdgperson_nodup ;
 by hh member;
run;
```

Finally, we should re-run the duplicate check to be sure there are no duplicates.

10.2. Spss

There is a menu for duplicate checking in Spss, however it creates rather complex syntax. Instead, we can write our own, easier syntax:

```
GET FILE='H:\MDG\Data\mdgperson.sav'.
SORT CASES BY hh member.
MATCH FILES FILE=*
            /FIRST=first_mem
            /LAST=last_mem
            /BY hh member
            .
EXECUTE.
TEMPORARY.
SELECT IF NOT(first_mem and last_mem).
LIST hh state TO last_mem.
```

This syntax will give us the same list as the Sas program did. When we want to delete true duplicates, we first sort the dataset by all the variables (there is a limit 64 variables, if we have more, we can sort by the 64 variables most likely to be different). Then we check for duplicates with the *Match files* command and select those who are the first within the group.

```
SORT CASES BY hh TO b6.
MATCH FILES FILE=*
            /FIRST=first_rec
            /LAST=last_rec
            /BY hh TO b6
            .
EXECUTE.
```

```
TEMPORARY.
SELECT IF NOT(first_rec and last_rec).
LIST hh TO last_rec.
SELECT IF (first_rec = 1).
EXECUTE.
```

The Spss log does not tell us if any observations were dropped.

For the duplicates with the wrong household number, we change the number for the first occurrence of each person with the number 040024 to 040034:

```
IF (hh = '040024' and first_mem = 1) hh = '040034'.
EXECUTE.
SORT CASES BY hh member.
```

Finally, we delete the first- and last-variables.

```
DELETE VARIABLES first_mem last_mem first_rec last_rec.
EXECUTE.
```

10.3. Stata

We sort the data by the identification variables. Then we generate a duplicate checking variable called *dup*. It will contain the number of records in each group. If this variable has a value greater than 1, the observations are duplicates on the identification.

```
use "h:\mdg\data\mdgperson.dta", clear
sort hh member, stable
by hh member:  generate dup = _N
list if dup > 1
```

Now we want to delete the true duplicates. First, we drop the variable *dup*. Then we sort on all variables and create a duplicate identifier called *dup_rec*. If it has a value above 1 it is the second or more duplicate and shall be deleted:

```
drop dup
sort _all, stable
by _all:  generate dup_rec = _N
drop if dup_rec>1
drop dup_rec
```

Stata tells us how many observations were dropped:

```
(1 observation deleted)
```

We now will want to change the value of household number from 040024 to 040034 as we did with Sas and Spss above. First, we must create the *dup* variable, with the syntax below it will contain a counter within each group. Then we do our corrections:

```
sort hh member, stable
by hh member:  generate dup = cond(_N==1,0,_n)
replace hh = "040034" if hh == "040024" & dup == 1
sort hh member
drop dup
```

Stata tells us how many changes were made:

```
(6 real changes made)
```

10.4.R

To delete rows that are exactly equal we can use the *duplicated* function. However when we want to remove duplicates we use the negation ! in front of duplicated:

```
mdgperson_noduprec <- mdgperson[!duplicated(mdgperson), ]
```

The syntax is somehow different from previous syntax. Within the brackets we define what to extract from the two dimensions of the data frame. Before the comma we select our rows and with !duplicated all rows that are not duplicates will be selected. We leave it empty after the comma and this means all columns shall be selected. The row names (or numbers) of the selected rows will not be updated. We can update them with this command:

```
rownames(mdgperson_noduprec)<-1:nrow(mdgperson_noduprec)
```

To list duplicates by identification variables, we can use the *group_by* in the *dplyr* package where we count the number of rows for the id variables and filter those who have more than one:

```
mdgperson_noduprec %>%
   group_by(hh,member) %>%
   add_count() %>%
   filter(n>1) %>%
   arrange(hh,member)
```

The code above gives us this list of duplicates:

```
   hh      state urbrur member    b3     b4     b5     b6      n

   <chr>   <chr> <chr>    <dbl>  <dbl>  <dbl>  <dbl>  <dbl>  <int>

 1 040024  4     1            1      0      2     20      3      2
 2 040024  4     1            1      0      1     37      2      2
 3 040024  4     1            2      1      2     33      2      2
 4 040024  4     1            2     11      2     23      3      2
 5 040024  4     1            3     11      2      7     NA      2
 6 040024  4     1            3      2      2     17      1      2
 7 040024  4     1            4      2      1     14      1      2
 8 040024  4     1            4     10      2      9     NA      2
 9 040024  4     1            5     10      2     18      1      2
10 040024  4     1            5      2      1      9     NA      2
11 040024  4     1            6      2      2     11     NA      2
12 040024  4     1            6     10      1     12      1      2
```

To change the values for the first of a duplicated household we can use *dplyr*. We start with grouping by hh and member and then we create a variable called member_first. This variable will have the value 1 when it is first within the group and 0 otherwise. Then we group by member to ungroup hh because we can't change a value that is defined in a group. We change to another hh number when it is the first within hh 040024. Finally, we sort the data with the *arrange* command and remove the variable member_first with the *subset* command. All these commands are piped together with *%>%*.

```
mdgperson_nodup <- mdgperson_noduprec %>%
   group_by(hh,member) %>%
   mutate(member_first = as.integer(ifelse((row_number() == 1), 1, 0))) %>%
   group_by(member) %>%
   mutate(hh=ifelse(hh=="040024" & member_first == 1,"040034",hh)) %>%
   arrange(hh,member) %>%
   subset(select=-member_first)
```

10.5. Python

Python can check if a record is a duplicate and can also choose which duplicate to delete. For a duplicate record we can delete duplicates like this:

```
mdgperson_noduprec = mdgperson.drop_duplicates()
```

As the rows for these duplicates are exactly equal, it does not matter if we keep the first or last of these rows. The intention is to get rid of all but one of the rows that are equal. That leaves us with a data frame without true duplicates.

Beware that the index (row numbers) is not updated. If we want to update the index, we can add the *ignore_index* option with the True parameter:

```
mdgperson_noduprec = mdgperson.drop_duplicates(ignore_index=True)
```

If we want to find the duplicates before we delete them, we can use the *duplicated* method:

```
mdgperson.loc[mdgperson.duplicated() == True]
```

To find the duplicates for identification columns we can also use the *duplicated* method, we just have to add the identification columns:

```
mdgperson_noduprec[mdgperson_noduprec.duplicated(['hh', 'member'],
keep='first')].sort_values(['hh', 'member'])
```

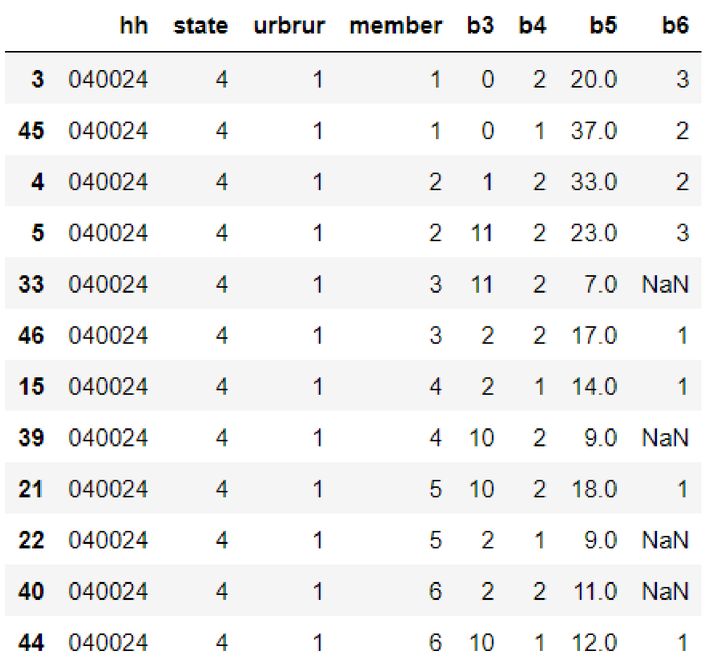
This gives us a list of the last occurrences of duplicates. The *keep=first* option tells us that the first occurrence is to be kept, hence the last will be the duplicate:

	hh	state	urbrur	member	b3	b4	b5	b6
45	040024	4	1	1	0	1	37.0	2
5	040024	4	1	2	11	2	23.0	3
46	040024	4	1	3	2	2	17.0	1
39	040024	4	1	4	10	2	9.0	NaN
22	040024	4	1	5	2	1	9.0	NaN
44	040024	4	1	6	10	1	12.0	1

If we want a list of all duplicates, we can use *keep=False* which selects all duplicates:

```
mdgperson_noduprec[mdgperson_noduprec.duplicated(['hh', 'member'],
keep=False)].sort_values(['hh', 'member'])
```

This will list all the rows with common combinations of the columns *hh* and *member*:

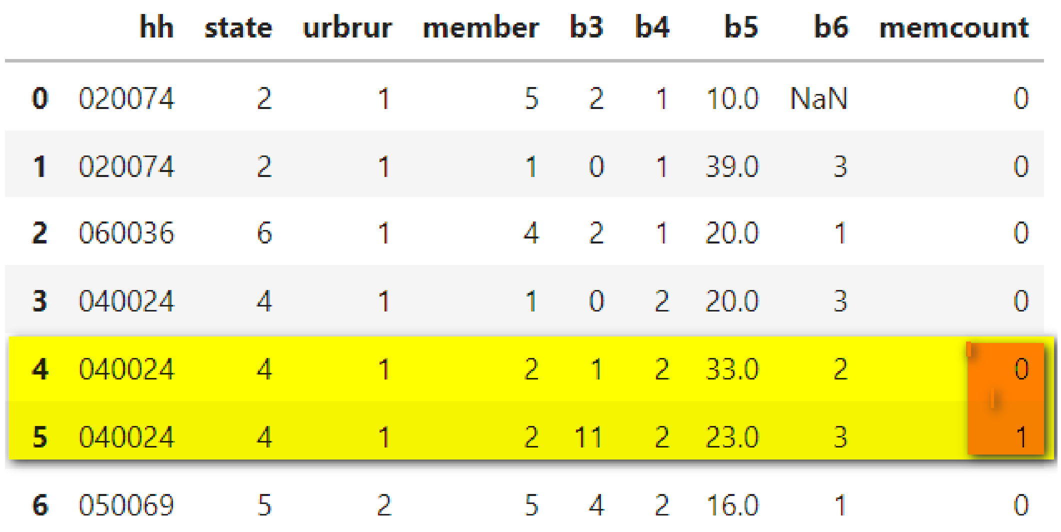


	hh	state	urbrur	member	b3	b4	b5	b6
3	040024	4	1	1	0	2	20.0	3
45	040024	4	1	1	0	1	37.0	2
4	040024	4	1	2	1	2	33.0	2
5	040024	4	1	2	11	2	23.0	3
33	040024	4	1	3	11	2	7.0	NaN
46	040024	4	1	3	2	2	17.0	1
15	040024	4	1	4	2	1	14.0	1
39	040024	4	1	4	10	2	9.0	NaN
21	040024	4	1	5	10	2	18.0	1
22	040024	4	1	5	2	1	9.0	NaN
40	040024	4	1	6	2	2	11.0	NaN
44	040024	4	1	6	10	1	12.0	1

We found out that the last of these duplicates should have the hh number 040034 instead of 040024. To change the hh for these duplicates we can first create a member count (memcount) for each combination of hh and member. The count will be 0 for the first, 1 for the second and so on. Then we can recode those with memcount = 1 using the *loc* method.

```
mdgperson_noduprec['memcount'] = mdgperson_noduprec.groupby(['hh',
'member']).cumcount()
mdgperson_noduprec.loc[(mdgperson_noduprec.memcount == 1) &
(mdgperson_noduprec.hh == '040024'), 'hh'] = '040034'
mdgperson_nodup = mdgperson_noduprec
mdgperson_nodup
```

Here we see in the memcount column that the first row within the group is 0 and the next is 1. In this way we can separate between the duplicates:



	hh	state	urbrur	member	b3	b4	b5	b6	memcount
0	020074	2	1	5	2	1	10.0	NaN	0
1	020074	2	1	1	0	1	39.0	3	0
2	060036	6	1	4	2	1	20.0	1	0
3	040024	4	1	1	0	2	20.0	3	0
4	040024	4	1	2	1	2	33.0	2	0
5	040024	4	1	2	11	2	23.0	3	1
6	050069	5	2	5	4	2	16.0	1	0

11. Labels for variables and data values

The variable names often do not describe the content of a variable very well. To add some more information about the content we should use labels. Adding variable labels will make it easier to use the dataset because it gives us a better documentation than just the variable names. Value labels (called formats in Sas) are also very helpful. It is much easier for humans to relate to a text instead of a code. Value labels are used to replace codes with texts.

11.1. Sas

We use the *Label* statement to add variable labels. This is normally done during a *Data* step. We should add variable labels to all variables we want to keep. When we import an external file to Sas we add variable labels. Each new variable we add later to datasets should also be labelled. Here is the import program used earlier, now with labels added:

```
DATA mdgperson ;
  INFILE 'H:\MDG\Data\mdgperson.txt' TRUNCOVER LRECL=15;
  INPUT
    @01 hh      $CHAR6.
    @07 state   1.
    @08 urbrur  1.
    @09 member  1.
    @10 b3      2.
    @12 b4      1.
    @13 b5      2.
    @15 b6      1.
    ;
  LABEL
    hh      = 'Household identification'
    state   = 'State'
    urbrur  = 'Urban/Rural location of household'
    member  = 'Member number within household'
    b3      = 'Relationship to head of household'
    b4      = 'Sex'
    b5      = 'Age'
    b6      = 'Civil status'
    ;
RUN;
```

Adding the value labels (formats) is done in a two-step process. First, we create the formats and then we connect the formats to their variables. The first step, creating the formats, looks like this:

```
PROC FORMAT;
 VALUE STATE (notsorted)
   1      = '01 Central'
   2      = '02 Capital'
   3      = '03 North'
   4      = '04 East'
   5      = '05 South'
   6      = '06 West'
   OTHER = 'N/A'
   ;
 VALUE URBRUR (notsorted)
   1      = 'Urban'
   2      = 'Rural'
   OTHER = 'N/A'
   ;
 VALUE HEAD_REL (notsorted)
   0      = 'Head'
   1      = 'Spouse'
   2      = 'Daughter/son'
   3      = 'Spouse of son/daughter'
   4      = 'Grandchild'
   5      = 'Sister/brother'
   6      = 'Sister/brother in-laws'
   7      = 'Parent'
   8      = 'Parent-in-law'
   9      = 'Niece/nephew'
   10     = 'Other relative'
   11     = 'Non relative'
   OTHER = 'N/A'
   ;
 VALUE SEX (notsorted)
   1      = 'Male'
   2      = 'Female'
   OTHER = 'N/A'
   ;
 VALUE CIVIL_STATUS (notsorted)
   1      = 'Never married'
   2      = 'Married - monogamy'
   3      = 'Married - polygamy'
   4      = 'Widowed'
   5      = 'Separated'
   6      = 'Divorced'
   .U     = 'N/A'
   OTHER = 'Missing'
   ;
RUN;
```

There is a menu in Enterprise guide for creating formats, however it is much easier to type them in directly as syntax in a program. It is also possible to create formats from Sas datasets.

The connection can be done in a *Data* step if we want a permanent connection or in listing procedures like *Proc freq, Proc means, Proc print* and so on if we want a temporary connection. Of course, the formats must be available before we connect them to variables. Here is the permanent connection added to the import program:

```
DATA mdgperson ;
  INFILE 'H:\MDG\Data\mdgperson.txt' TRUNCOVER LRECL=15;
  INPUT
    @01 hh      $CHAR6.
    @07 state   1.
    @08 urbrur 1.
    @09 member 1.
    @10 b3      2.
    @12 b4      1.
    @13 b5      2.
    @15 b6      1.
    ;
  LABEL
    hh      = 'Household identification'
    state   = 'State'
    urbrur = 'Urban/Rural location of household'
    member = 'Member number within household'
    b3      = 'Relationship to head of household'
    b4      = 'Sex'
    b5      = 'Age'
    b6      = 'Civil status'
    ;
  FORMAT
```

```
   state   STATE.
   urbrur URBRUR.
   b3      HEAD_REL.
   b4      SEX.
   b6      CIVIL_STATUS.  ;
RUN;
```

When we connect the formats to variables we must add a dot at the end of their names, as seen above. Format names for character variables must start with a $ sign. Numeric formats can only be used for numeric variables and vice versa.

11.2. Spss

It is possible to add variable and value labels from the variable view window. This is not recommended because we can't paste the syntax for these operations. If we then must re-create the dataset (which is often the case) all our labels will be gone, and we have to type in the labels once more. It is far better to use syntax to add the labels. The labels will be permanently added to the active dataset when the commands are executed. Here is the syntax for variable labels:

```
VAR LABELS
    hh      'Household identification'
    state   'State'
    urbrur  'Urban/Rural location of household'
    member  'Member number within household'
    b3      'Relationship to head of household'
    b4      'Sex'
    b5      'Age'
    b6      'Civil status'
    .
```

For value labels it is done in one step. When we create the labels, we also tell Spss which variables they shall connect to:

```
VALUE LABELS
    state
            1   '01 Central'
            2   '02 Capital'
            3   '03 North'
            4   '04 East'
            5   '05 South'
            6   '06 West'
    /
    urbrur
            1   'Urban'
            2   'Rural'
    /
    b3
            0   'Head'
            1   'Spouse'
            2   'Daughter/son'
            3   'Spouse of son/daughter'
            4   'Grandchild'
            5   'Sister/brother'
            6   'Sister/brother in-laws'
            7   'Parent'
            8   'Parent-in-law'
            9   'Niece/nephew'
            10   'Other relative'
            11   'Non relative'
    /
    b4
            1  'Male'
            2  'Female'
    /
    b6
            1  'Never married'
            2  'Married - monogamy'
            3  'Married - polygamy'
            4  'Widowed'
            5  'Separated'
            6  'Divorced'
            9  'N/A'
    .
```

If the same value labels shall be used for more variables, we don't repeat the whole value labels command. We just add more variable names to the same value labels.

11.3. Stata

The variable labels are permanently connected to the opened dataset, similar to Spss:

```
label variable hh      "Household identification"
label variable state   "State"
label variable urbrur  "Urban/Rural location of household"
label variable member  "Member number within household"
label variable b3      "Relationship to head of household"
label variable b4      "Sex"
label variable b5      "Age"
label variable b6      "Civil status"
```

Beware that only numeric variables may be given value labels. They are added in two steps, like it is done in Sas. First, we define the value labels:

```
label define state ///
1 "01 Central" ///
2 "02 Capital" ///
3 "03 North" ///
4 "04 East" ///
5 "05 South" ///
6 "06 West"
label define urbrur ///
1 "Urban" ///
2 "Rural"
label define head_rel ///
0 "Head" ///
1 "Spouse" ///
2 "Daughter/son" ///
3 "Spouse of son/daughter" ///
4 "Grandchild" ///
5 "Sister/brother" ///
6 "Sister/brother in-laws" ///
7 "Parent" ///
8 "Parent-in-law" ///
9 "Niece/nephew" ///
10 "Other relative" ///
11 "Non relative"
label define sex ///
1 "Male" ///
2 "Female"
```

```
label define civil_status ///
1 "Never married" ///
2 "Married - monogamy" ///
3 "Married - polygamy" ///
4 "Widowed" ///
5 "Separated" ///
6 "Divorced" ///
.u "N/A"
```

Next, we add the value labels permanently to their variables:

```
label values state     state
label values urbrur    urbrur
label values b3        head_rel
label values b4        sex
label values b6        civil_status
```

11.4.R

We add variable labels to a data frame by first creating a vector with the variable names and labels. Then we add them to the data frame with the *upData* command in the *Hmisc* package. As the *Hmisc* package is not attached as default we attach it first with the *library* command:

```
mdgperson.var.labels <- c(hh     = 'Household identification',
                          state  = 'State',
                          urbrur = 'Urban/Rural location of household',
                          member = 'Member number within household',
                          b3     = 'Relationship to head of household',
                          b4     = 'Sex',
                          b5     = 'Age',
                          b6     = 'Civil status'
                          )
library(Hmisc)
mdgperson_nodup <- upData(mdgperson_nodup, labels = mdgperson.var.labels)
```

In R we don’t use value labels (formats) in the same way as in the other packages described in this document. Instead, R has its own variable type called factor. When we define a variable as a factor it converts the values to indexes which are called value levels. These indexes may be given texts, called value labels.

We can see this with a simple example. First, we create a vector with values for sex:

```
sex <- c("m","k","k","m","k")
str(sex)
```

The structure looks like this:

```
chr [1:5] "m" "k" "k" "m" "k"
```

We can create a factor from the vector:

```
sexf <- factor(sex)
str(sexf)
```

The structure now looks like this:

```
Factor w/ 2 levels "k","m": 2 1 1 2 1
```

The factor has two levels (unique values), k and m. They are represented with index values, 1 for females (k) and 2 for males (m).

To give texts to the values of sex we can convert to factor and at the same time define the levels with its labels:

```
sexfl <- factor(sex,levels=c("m","k"),labels=c("Male","Female"))
str(sexfl)
```

Males are given the index 1 and females 2 because we defined male before female:

```
Factor w/ 2 levels "Male","Female": 1 2 2 1 2
```

This is opposite of the first example where females got 1 and males 2. The index always starts with 1 and for each unique value it is added with 1. For values that goes from 1 and increase by one for each value the index will have the same value as the original values. Otherwise, the index will be different from the original value.

When a variable is defined as a factor and we want to subset on values for the variable, we should use the variable labels, not the original values. We can use the index, but then we must know which index value each unique value has. Some examples with their results, we see that the indexes are different between the *sexf* and the *sexfl* variables:

```
> subset(sex,sex =="k")

[1] "k" "k" "k"

> subset(sexf,sexf =="k")

[1] k k k

Levels: k m

> subset(sexf,as.numeric(sexf) =="1")

[1] k k k

Levels: k m

> subset(sexfl,sexfl =="Female")

[1] Female Female Female

Levels: Male Female

> subset(sexfl,as.numeric(sexfl) =="2")

[1] Female Female Female

Levels: Male Female
```

Now we can change our categorical variables to factors. We start with defining the values (levels) and labels for each variable and put them into separate vectors:

```
state.codes <- c(1,2,3,4,5,6)
state.texts <- c('01 Central','02 Capital','03 North','04 East','05 South','0
6 West')
urbrur.codes <- c(1,2)
urbrur.texts <- c('Urban','Rural')
head_rel.codes <- c(0,1,2,3,4,5,6,7,8,9,10,11)
head_rel.texts <- c("Head","Spouse","Daughter/son","Spouse of son/daughter","
Grandchild","Sister/brother","Sister/brother in-laws","Parent","Parent-in-law
","Niece/nephew","Other relative","Non relative")
sex.codes <- c(1,2)
sex.texts <- c('Male','Female')
```

```
civil_status.codes <- c(1,2,3,4,5,6)
civil_status.texts <- c('Never married','Married - monogamy','Married - polyg
amy','Widowed','Separated','Divorced')
```

We can now use these vectors to define the values and labels for each factor variable we create:

```
mdgperson_nodup$state <- factor(mdgperson_nodup$state,levels = state.codes,la
bels=state.texts)
mdgperson_nodup$urbrur <- factor(mdgperson_nodup$urbrur,levels = urbrur.codes
,labels=urbrur.texts)
mdgperson_nodup$b3 <- factor(mdgperson_nodup$b3,levels = head_rel.codes,label
s=head_rel.texts)
mdgperson_nodup$b4 <- factor(mdgperson_nodup$b4,levels = sex.codes,labels=sex
.texts)
mdgperson_nodup$b6 <- factor(mdgperson_nodup$b6,levels = civil_status.codes,l
abels=civil_status.texts)
```

To see the different texts for a variable we can use the *levels* command:

```
levels(mdgperson_nodup$b6)
```

The levels are as follows:

```
[1] "Never married"      "Married - monogamy" "Married - polygamy"

[4] "Widowed"            "Separated"          "Divorced"
```

11.5. Python

We can add variable labels as attributes to a data frame. However, they are not shown in any outputs. What we can do, though, is to list these attributes. Here is an example:

```
varlabels = {'hh': 'Household number',
             'state': 'State',
             'urbrur': 'Urban/rural',
             'member': 'Member number',
             'b3': 'Relationshipp to head',
             'b4': 'Sex',
             'b5': 'Age',
             'b6': 'Civil status'
            }
mdgperson_nodup.attrs = varlabels
mdgperson_nodup.attrs
```

This will give us a listing like this:

```
{'hh': 'Household number',
 'state': 'State',
 'urbrur': 'Urban/rural',
 'member': 'Member number',
 'b3': 'Relationshipp to head',
 'b4': 'Sex',
 'b5': 'Age',
 'b6': 'Civil status'}
```

We can also list just one variable label:

```
mdgperson_nodup.attrs['b5']
```

When it comes to value labels for variables, we can put the codes and texts in a nested dictionary and then add them to our data frame with a replace command. Here is an example on how a nested dictionary with value labels for several variables may look like:

```
labels = {'state':
          {'1': '01 Central',
           '2': '02 Capital',
           '3': '03 North',
           '4': '04 East',
           '5': '05 South',
           '6': '06 West',
           np.nan: 'missing'},
          'urbrur':
          {'1': 'Urban',
           '2': 'Rural'},
          'b3':
          {'0': 'Head',
           '1': 'Spouse',
           '2': 'Daughter/son',
           '3': 'Spouse of son/daughter',
           '4': 'Grandchild',
           '5': 'Sister/brother',
           '6': 'Sister/brother in-laws',
           '7': 'Parent',
           '8': 'Parent-in-law',
           '9': 'Niece/nephew',
           '10': 'Other relative',
           '11': 'Non relative'},
          'b4':
          {'1': 'Male',
           '2': 'Female'},
          'b6':
          {'1': 'Never married',
           '2': 'Married - monogamy',
           '3': 'Married - polygamy',
           '4': 'Widowed',
           '5': 'Separated',
           '6': 'Divorced',
           np.nan: 'missing'}
         }
```

The variable names (state, urbrur etc.) are used as dictionary keys. For each key we have sets consisting of codes and texts. That is what makes the directories nested. The codes must be of the same types as they are in the data frame. If it is to be connected to an object variable, the codes should be in quotes, otherwise no quotes are needed. Beware that you might have to add blanks if codes are of different lengths (see the *b3* variable it has codes with both 1 and 2 digits. For those with one digit, a space has been added at the beginning).

Now that we have a dictionary with our code lists, we want to add them to our data frame. It can be done separately for each variable, for instance with the *map* function:

```
mdgperson_nodup['state'] = mdgperson_nodup['state'].map(labels['state'])
```

When we have many code lists to add, it is better to add them all in one command. We do this with the *replace* function:

```
mdgperson_nodup = mdgperson_nodup.replace(labels)
```

If the labels are in different nested directories, we can add them together with a concatenate directories syntax: {**dict1, **dict2}. Here is an example:

```
states = {'state':
          {'1': '01 Central',
           '2': '02 Capital',
           '3': '03 North',
           '4': '04 East',
           '5': '05 South',
           '6': '06 West',
           np.nan: 'missing'}
         }
urbanity = {'urbrur':
            {'1': 'Urban',
             '2': 'Rural'}
           }
labs = {**states, **urbanity}
labs
```

12. Storing datasets

When we have processed our data, we usually want to store them permanently. If we have not stored our datasets, both Spss and Stata ask us if we want to store when we exit. R will also ask if we want to store unsaved items. Nevertheless, we should store our data before we exit. In Sas we must make sure to store our data permanently. The default is that all datasets are stored in a temporary folder which is deleted when we exit Sas. To store datasets permanently we first make a Sas alias for the folder in which we will store. This alias is called a *libref* and is defined with a *Libname* statement. We don't have a specific command for storing datasets permanently, we decide if the data is to be stored temporary or permanently when we name our output dataset in a *Data step* or in a *Proc step*.

In Spss we use the *Save* command and use the whole physical file name with the .sav extension. There is an option to define the name of the working directory with the *Cd* command. If we use this, we can later omit the path in the *Save* command, as it has already been defined. For Stata we use a *Save* command as well. We can also use a *Cd* command similar to Spss to define the working directory. For R we can use the *save* or *saveRDS* commands, they store the data in two different formats where RDS is the latest. In Python we can save in various data formats. However, the parquet format seems to be the preferred one.

12.1. Sas

We must define a folder name in Sas for the permanent storage of our data. This is done with the *libname* statement, and the folder is given a local name in Sas, an alias. The folder h:\mdg\data is called mdg in Sas. This is called a libref. After the *libname* statement is executed, Sas will now which folder to search in every time we reference to the libref mdg:

```
libname mdg 'H:\MDG\Data';
proc sort data=mdgperson_nodup out=mdg.mdgperson_nodup;
 by hh member;
run;
```

The input dataset in this sort, *mdgperson_nodup* is temporary while the output dataset *mdg. mdgperson_nodup* will be permanent. When we omit the *libref*, Sas uses the dataset with the *libref* for the temporary folder *work*. If we try to read or write datasets from a permanent directory and it is not defined in the Sas session, we will get an error message like this:

```
ERROR: Libname MDG is not assigned.
```

This means that the *Libname* statement must be executed every time a Sas session is started for the folder to be available for our use in Sas.

Formats may also be stored permanently. We must add a folder for the format file with another *Libname* statement and then add the option *Libname* to the *Proc format* statement. The formats will be stored in a file called formats.sas7bcat. The *libref* for the formats must be called *library*. It may look like this:

```
libname library 'H:\MDG\Cat';
PROC FORMAT LIB=library;
 VALUE SEX (notsorted)
   1     = 'Male'
   2     = 'Female'
   OTHER = 'N/A'
   ;
RUN;
```

12.2. Spss

To store the active dataset, we simply use the *Save* command:

```
SAVE /OUTFILE 'h:\mdg\data\mdgperson_nodup.sav' .
```

As mentioned above we can extract the path into a *Cd* command and then use only the filename in the *Save* command:

```
CD 'h:\mdg\data'.
SAVE /OUTFILE 'mdgperson_nodup.sav' .
```

12.3. Stata

Stata is similar to Spss, we use a *save* command. Replace is not the default here as it is in Spss, so we add the replace option:

```
save "h:\mdg\data\mdgperson_nodup.dta", replace
```

If we use the *cd* command, it will be like this:

```
cd "h:\mdg\data\"
save "mdgperson_nodup.dta", replace
```

Using these relative paths makes it easier to move or copy our Spss and Stata systems. If we want to move a system from a path called h:\mdg\data to c:\mdg\data we will need to change the path only in the *cd* command. If we don't use relative paths, we will have to change all places where a path is mentioned. However, it does not help much to add the *cd* command to every syntax file. We should extract it to a file that could be invoked at start-up. The next chapter describes how we do this.

12.4. R

We can use either the *save* or the *saveRDS* commands to save data files in R format. With the save command the data file will be saved as a Rdata file, hence we should add the extension .Rdata:

```
save(mdgperson_nodup,file="h:/MDG/Data/mdgperson_nodup.Rdata")
```

When we want to save the file as an RDS file, we use the .rds extension:

```
saveRDS(mdgperson_nodup,file="h:/MDG/Data/mdgperson_nodup.rds")
```

We can set the working directory and then use relative paths instead of the full path name. To check what the current working directory is we use the the getwd command:

```
getwd()
```

To change the working directory, we use the setwd command:

```
setwd("h:/MDG")
```

Now we can save the file in the Data folder in H:/MDG without typing the full path:

```
save(mdgperson_nodup,file="Data/mdgperson_nodup.Rdata")
saveRDS(mdgperson_nodup,file="Data/mdgperson_nodup.rds")
```

12.5. Python

We can store data frames as python pickles, which have the extension .pkl. It is an easy operation:

```
mdgperson_nodup.to_pickle(datapath + 'mdgperson_nodup.pkl')
```

To open a pickle, we can do this:

```
mdg_from_pickle = pd.read_pickle(datapath + 'mdgperson_nodup.pkl')
```

The pickle format is not safe as it may include malicious code which can be executed while reading.

Another possibility is to save our file in json format:

```
mdgperson_nodup.to_json(datapath + 'mdgperson_nodup.json')
```

This may be opened again like this:

```
mdg_from_json = pd.read_json(datapath + 'mdgperson_nodup.json')
```

As parquet is the recommended data storage for Python, we should use it. However, it is not included in the standard version of Python. Hence, we must install and import it. The installation can be done with this code (when we are connected to internet):

```
!pip install pyarrow
```

This is a one-time operation. The ! at the beginning tells Python to run a system command, which in this case is pip. pip is used to install additional modules in Python. Before we can save as a parquet file, we must import the pyarrow module, like this:

```
import pyarrow
```

Now we can save our data frame as a parquet file:

```
mdgperson_nodup.to_parquet(datapath + 'mdgperson_nodup.parquet')
```

Now we can open the parquet file again:

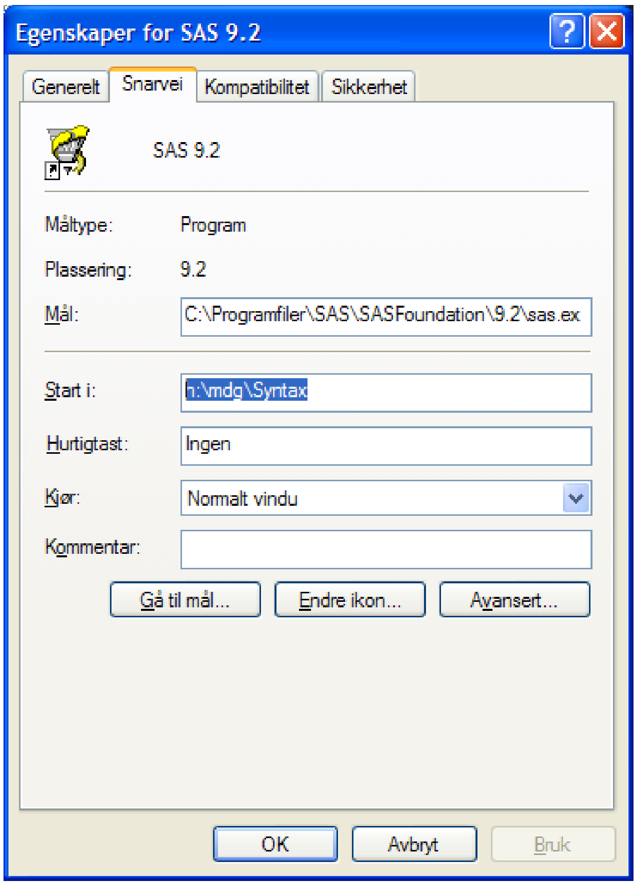
```
mdg_from_parquet = pd.read_parquet(datapath + 'mdgperson_nodup.parquet')
```

13. Automatic invocation at start-up

We usually want to execute some commands every time we start a session. When we are ready to work, all these commands should have been executed. This is possible in Sas and Stata. In Spss we can open a syntax file on invocation, but there is no easy way to execute it automatically. The commands we want to execute on start-up are typically definition of working directories and they will differ from one task to another (i.e., working separately on two different surveys). Therefore, we should have separate icons for each survey.

13.1. Sas

In Base Sas we put our start-up statements in a file called autoexec.sas. When this file is located in the start-up directory, the statements will be executed upon start. As there will be separate autoexec.sas files for each survey, they should have separate Sas icons. When a Sas icon is on our desktop, we will change the start-up directory in its properties:


Egenskaper for SAS 9.2

Generelt | Snarvei | Kompatibilitet | Sikkerhet

SAS 9.2

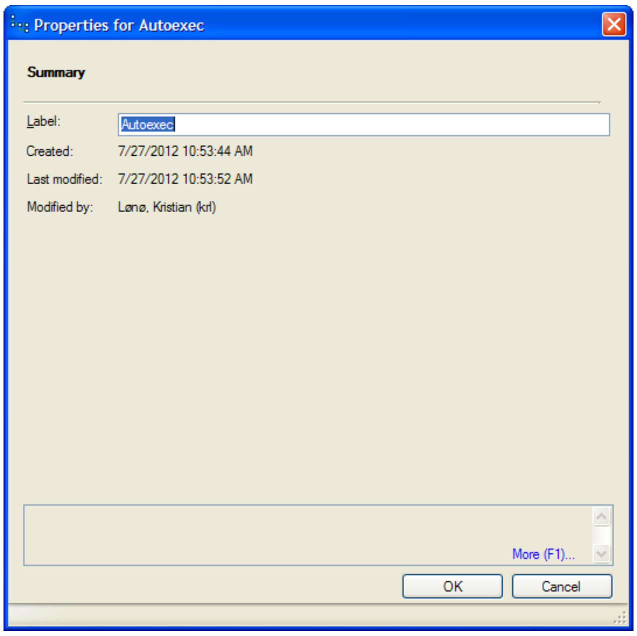
Måltype:	Program
Plassering:	9.2
Mål:	C:\Programfiler\SAS\SASFoundation\9.2\sas.ex
Start i:	h:\mdg\Syntax
Hurtigtast:	Ingen
Kjør:	Normalt vindu
Kommentar:	

Here we have changed the start directory to h:\mdg\syntax. If we put an autoexec.sas file in this folder the statements in that file will be executed upon start. The autoexec.sas may look like this:

```
libname mdg 'H:\MDG\Data';
libname library 'H:\MDG\Cat';
```

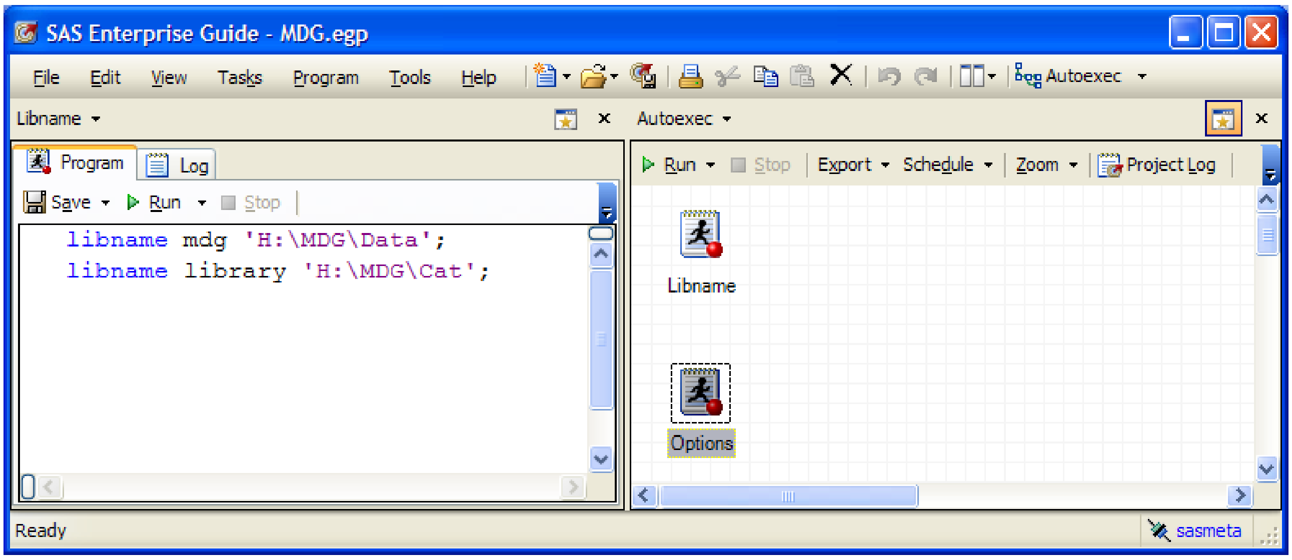
If we will use more permanent folders for our data, we simply add more *Libname* statements to the autoexec.sas file. If we have these *Libname* statements in other programs, we should remove them from the other programs when they are put in the autoexec.sas file. We can add other statements in the autoexec.sas file as well.

If we use Enterprise Guide, we will create a process flow called *Autoexec* and put the start-up tasks there. We add a new process flow with File > New > Process flow. We will rename this new process flow by opening the process flow properties and change the name:



Label: Autoexec
Created: 7/27/2012 10:53:44 AM
Last modified: 7/27/2012 10:53:52 AM
Modified by: Lønø, Kristian (krl)

We create a new program with File > New > Program and type in the *Libname* statement. It will appear like this in the process flow chart (shown side by side. Menu: View > Workspace layout > Side by side):


SAS Enterprise Guide - MDG.egp
File Edit View Tasks Program Tools Help
Autoexec
Libname
Program Log
Save Run Stop
Autoexec
Run Stop Export Schedule Zoom Project Log
Libname
Options
Ready
sasmeta

```
libname mdg 'H:\MDG\Data';
libname library 'H:\MDG\Cat';
```

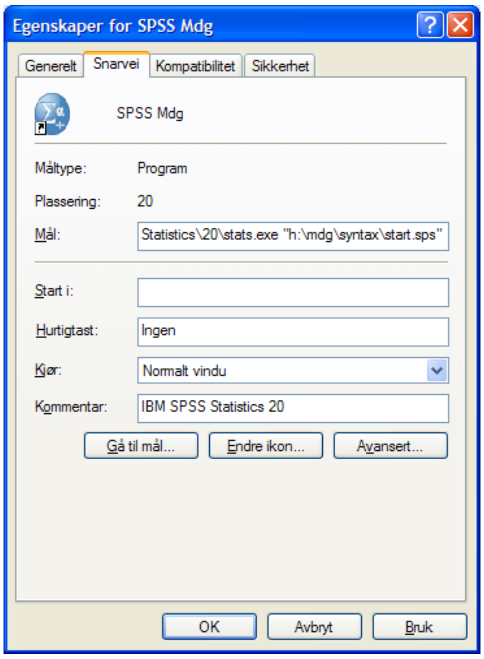
When we open this project the Autoexec process flow will be executed. As mentioned before *Libname* statements should only be placed in the autoexec.sas process flow.

13.2. Spss

There is no easy way to automate execution of commands at start-up in Spss. A solution is to open a syntax file when Spss starts and execute it ourselves. We move the *Cd* command to a file called start.sps which is to be located in the syntax folder of the survey:

```
CD 'h:\mdg\data'.
```

Then we will change the properties for the Spss icon for our survey on the desktop. At the invocation of Spss we add our start.sps command file:


Egenskaper for SPSS Mdg

Generelt | Snarvei | Kompatibilitet | Sikkerhet

SPSS Mdg

Måltype: Program
Plassering: 20
Mål: Statistics\20\stats.exe "h:\mdg\syntax\start.sps"
Start i:
Hurtigtast: Ingen
Kjør: Normalt vindu
Kommentar: IBM SPSS Statistics 20

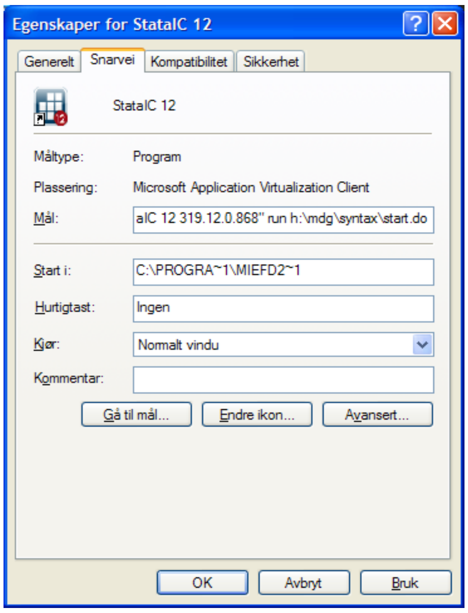
Gå til mål... | Endre ikon... | Avansert...

OK | Avbryt | Bruk

The syntax file h:\mdg\syntax\start.sps will now be opened when we start Spss.

13.3. Stata

In Stata we can run a command file upon invocation. We add the command run and the name of the start-up command file in the properties for the Stata Icon for our survey:


Egenskaper for StataIC 12

Generelt | Snarvei | Kompatibilitet | Sikkerhet

StataIC 12

Måltype: Program
Plassering: Microsoft Application Virtualization Client
Mål: aIC 12 319.12.0.868" run h:\mdg\syntax\start.do
Start i: C:\PROGRA~1\MIEFD2~1
Hurtigtast: Ingen
Kjør: Normalt vindu
Kommentar:

Gå til mål... | Endre ikon... | Avansert...

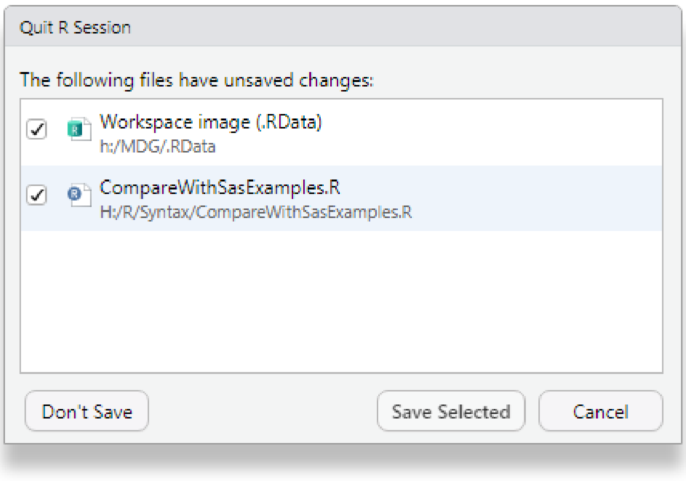
OK | Avbryt | Bruk

The command file h:\mdg\syntax\start.do will be executed when we start Stata. The start.do file looks like this:

```
cd "h:\mdg\data"
```

1.2R

In RStudio we can save our workspace when we exit. When we start Rstudio we will be back were we last ended. We just save the workspace when we exit:


Quit R Session

The following files have unsaved changes:

✓ Workspace image (.RData)
h:/MDG/.RData

✓ CompareWithSasExamples.R
H:/R/Syntax/CompareWithSasExamples.R

Libraries not attached in the basic R will have to attached with the *library* command, though.

13.4. Python

There are no built-in procedures for automatic execution of start-up code in Python. There are some techniques for automatic execution of code at session invocation. However, these are not discussed in this document.

14. Matching files

To match files together is a very important part of data processing. There are usually several ways to match datasets together. We must choose the right way according to the results we want. We also must know the algorithms of matching. The algorithm may be different from one software to another. We should not match more than two datasets at a time. Another thing to avoid is to have common variables in both datasets which are not part of the matching key. This usually causes trouble. We have added a common variable, *county*, in our example datasets to show how it is treated in the different matching.

When we match two datasets with a key variable it may appear:

- 1 time in each dataset (1:1)
- 1 time in dataset 1 and many times in dataset 2 (1:many)
- Many times in dataset 1 and 1 time in dataset 2 (many:1)
- Many times in dataset 1 and many times in dataset 2 (many:many)
- 1 time in dataset 1 and zero times in dataset 2 (1:zero)
- Many times in dataset 1 and zero times in dataset 2 (many:zero)
- Zero times in dataset 1 and 1 time in dataset 2 (zero:1)
- Zero times in dataset 1 and many times in dataset 2 (zero:many)

We will see how Sas, Spss, Stata, R and Python treat these situations by looking at match examples. We start with two data files with a common key variable. In our datasets all the above situations will appear when we match the datasets. Here are the two datasets:

Reg 1.

id mstat county

```
3 c 03
1 a
2 b 02
4 e 05
3 d 04
4 f 06
4 g 07
7 i 09
5 h 08
7 j 10
```

Reg 2.

id cstat county

```
2 y 19
1 z 20
2 x 18
4 v 06
3 w 17
4 u 15
8 s 12
6 t 14
8 r 11
```

We should be very careful to match datasets with duplicates on both files. Usually, we get rid of duplicates in at least one of the files before we match. However, sometimes we need to match datasets with duplicates and therefore we should know the algorithms for this kind of matching too.

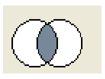
We will now look at some different matching we can do in each of the software packages.

14.1. Sas

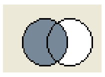
We start with importing the data files to Sas. In this example we include the file in the import program:

```
data reg1;
 input
  @1 id       $1.
  @3 mstat    $1.
  @5 county   $2.
  ;
cards;
3 c 03
1 a
2 b 02
4 e 05
3 d 04
4 f 06
4 g 07
7 i 09
5 h 08
7 j 10
;
run;
data reg2;
 input
  @1 id       $1.
  @3 cstat    $1.
  @5 county   $2.
  ;
cards;
2 y 19
1 z 20
2 x 18
4 v 06
3 w 17
4 u 15
8 s 12
6 t 14
8 r 11
;
run;
```

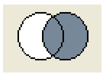
We can use *Proc sql, Data step* with *Merge* and *Data step* with *Update* to match datasets in Sas. We start with looking at *Proc sql*. *Proc sql* has many ways to match datasets; inner join, right join, left join and full join are the most commonly used. The difference between these is which observations are written to the output dataset. The match algorithm is the same.



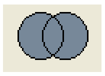
Inner join. Only observations with matching keys are written to output.



Left join. Observations with matching keys and observations only in the left (first mentioned) dataset are written to output.

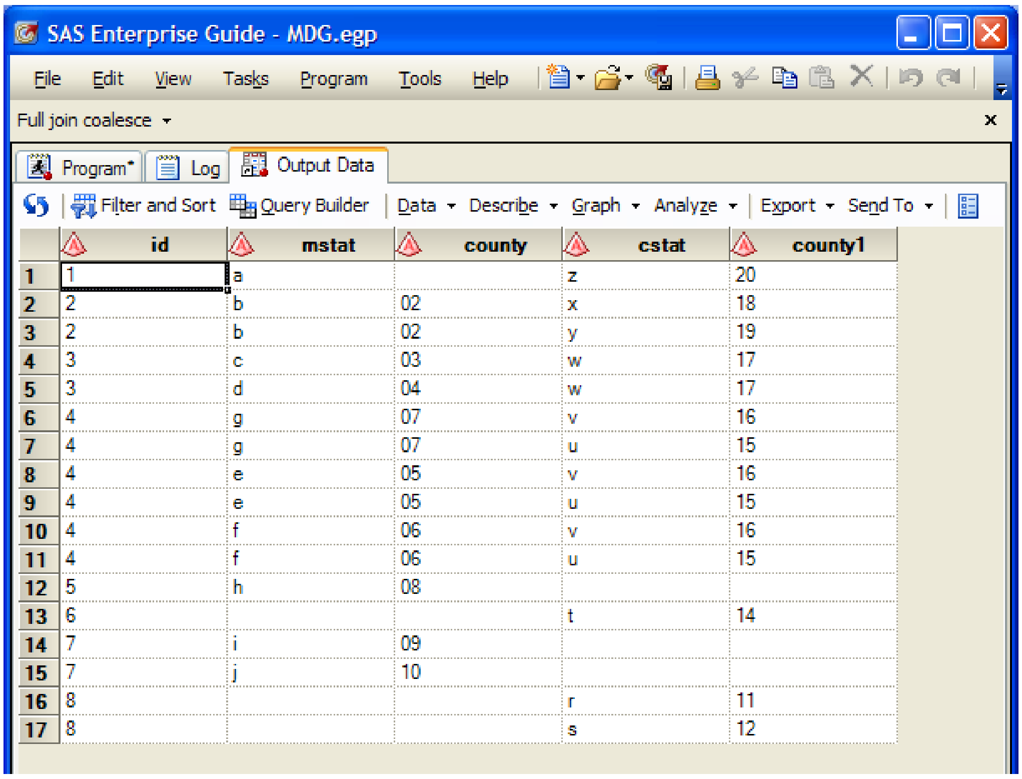


Right join. Observations with matching keys and observations only in the right (last mentioned) dataset are written to output.



Full join. All observations are written to output.

The algorithm of the match is that all observations with the same keys in both datasets match with each other. The result of full join shows this:


SAS Enterprise Guide - MDG.egp
File Edit View Tasks Program Tools Help
Full join coalesce
Program* Log Output Data
Filter and Sort Query Builder Data Describe Graph Analyze Export Send To

	id	mstat	county	cstat	county1
1	1	a		z	20
2	2	b	02	x	18
3	2	b	02	y	19
4	3	c	03	w	17
5	3	d	04	w	17
6	4	g	07	v	16
7	4	g	07	u	15
8	4	e	05	v	16
9	4	e	05	u	15
10	4	f	06	v	16
11	4	f	06	u	15
12	5	h	08		
13	6			t	14
14	7	i	09		
15	7	j	10		
16	8			r	11
17	8			s	12

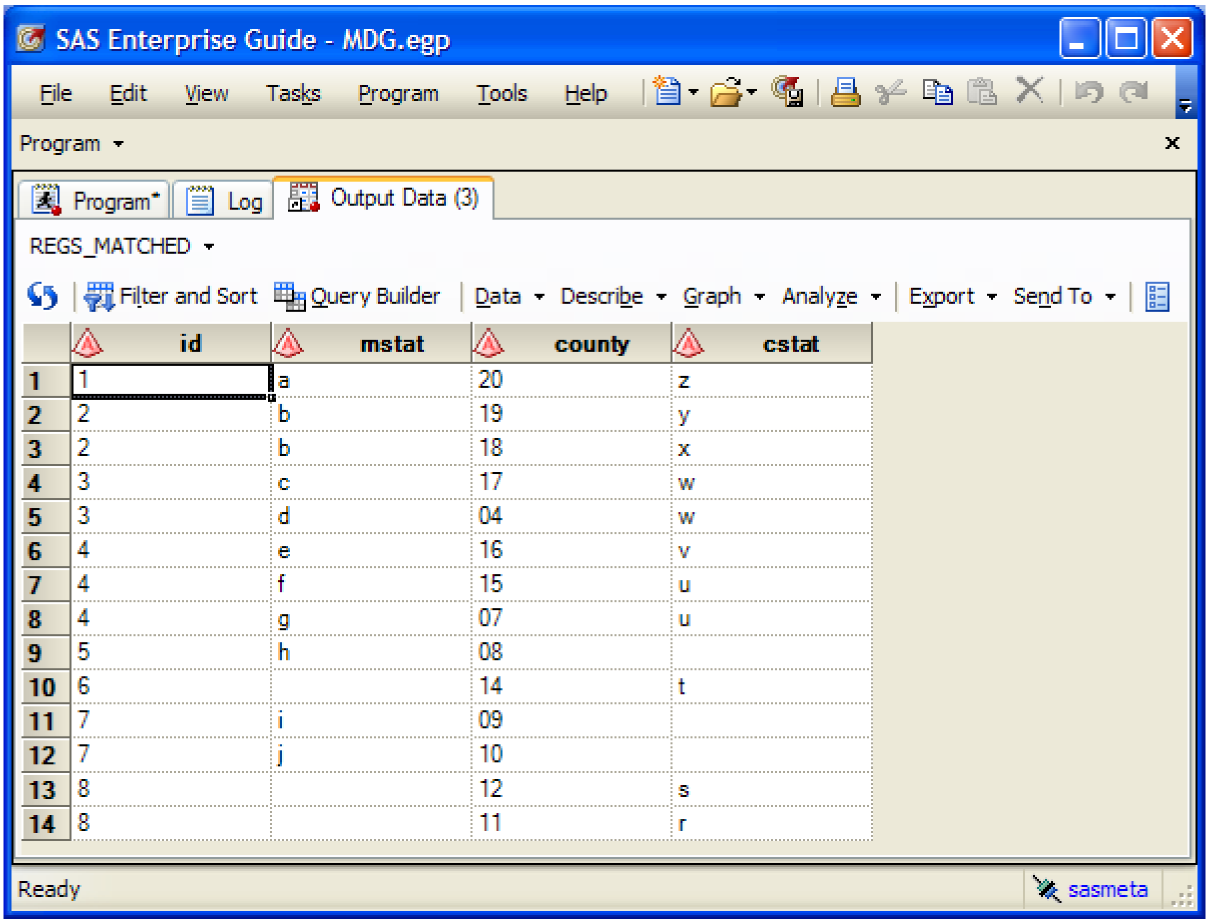
The syntax of the *Proc sql* looks like this:

```
PROC SQL;
   CREATE TABLE regs_joined2 AS
   SELECT coalesce(t1.id, t2.id) as id,
          t1.mstat,
          t1.county,
          t2.cstat,
          t2.county AS county1
      FROM WORK.REG1 t1 FULL JOIN WORK.REG2 t2 ON (t1.id = t2.id)
      ORDER BY id;
QUIT;
```

The *coalesce* function is used to unite the id variables from both datasets into one variable in the output dataset. For inner, right and left joins we do not need to use the *coalesce* function.

When we use the *Data step* to merge the two datasets they must be sorted on the key variable before the match is done. The default match is similar to the full join. However, the algorithm is different. Here is syntax for a merge:

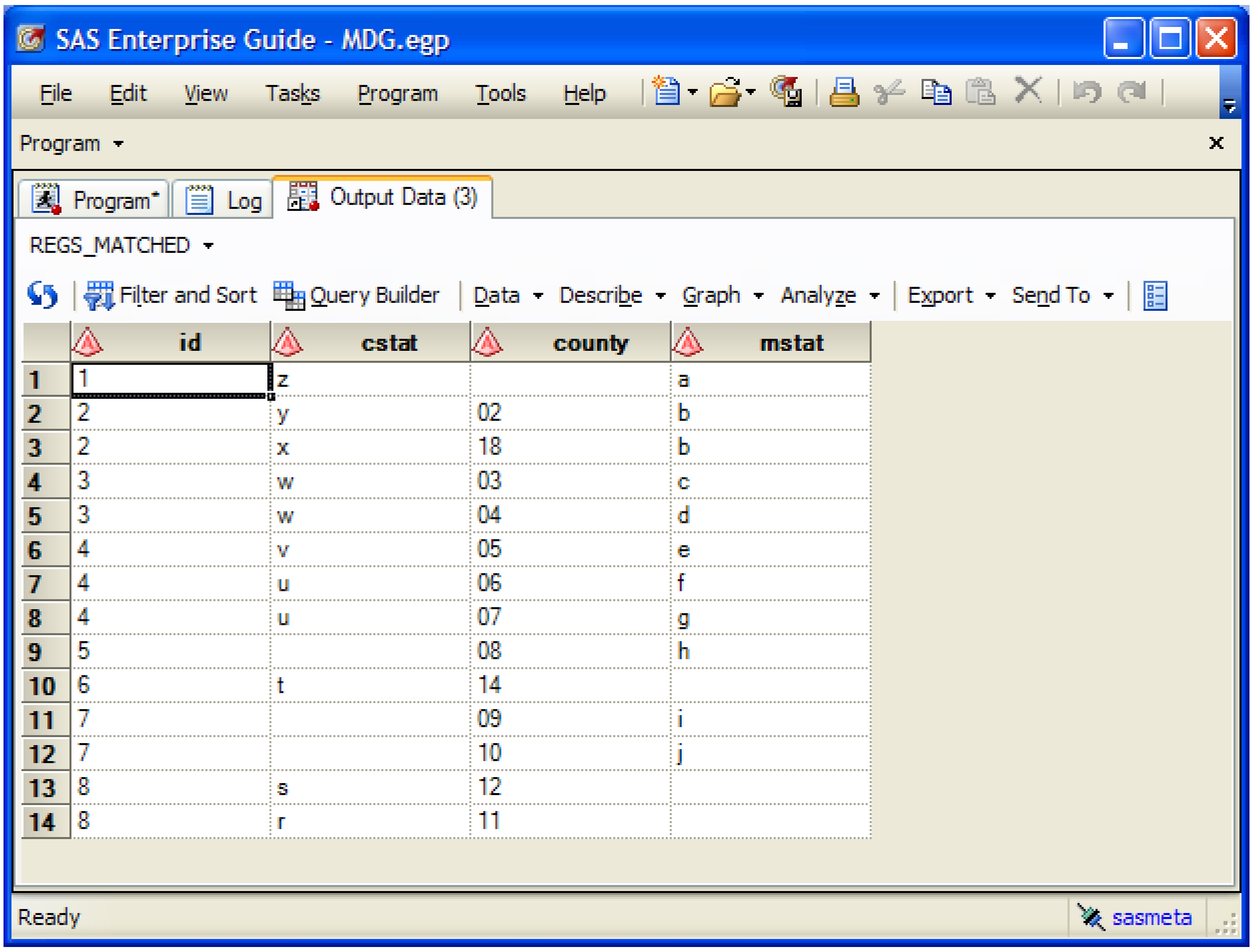
```
proc sort data=reg1 OUT=reg1s;
 by id;
run;

proc sort data=reg2 OUT=reg2s;
 by id;
run;

data regs_matched_all ;
   merge reg1s
         reg2s
         ;
   by id ;
run;
```


SAS Enterprise Guide - MDG.egp
File Edit View Tasks Program Tools Help
Program
Program* Log Output Data (3)
REGS_MATCHED
Filter and Sort Query Builder Data Describe Graph Analyze Export Send To
Ready
sasmeta

	id	mstat	county	cstat
1	1	a	20	z
2	2	b	19	y
3	2	b	18	x
4	3	c	17	w
5	3	d	04	w
6	4	e	16	v
7	4	f	15	u
8	4	g	07	u
9	5	h	08	
10	6		14	t
11	7	i	09	
12	7	j	10	
13	8		12	s
14	8		11	r

The algorithm difference between *Proc sql* and the *Data step* with *Merge* appears mainly when there are duplicates on both datasets, which happen for id = 4 in our example. While *Proc sql* matched all with the same keys in the first dataset with all with the same keys in the second, *Merge* will turn the many:many matches into as many 1:1 matches as possible and the remaining will be changed into either a 1:many or many:1 match. For 1:many and many:1 matches the one matches all in the other dataset. However, the order of the datasets mentioned in the *Merge* statement will influence the content of variables with common names which are not key variables in the datasets. This happens for the *county* variable in our example. The variable is in both datasets but not as a key variable. The variable will end up with data from the dataset that is read last and that differs when we swap the order in the *Merge* statement:

```
data regs_matched_all_opp ;
   merge reg2s
         reg1s
         ;
   by id ;
run;
```


SAS Enterprise Guide - MDG.egp
File Edit View Tasks Program Tools Help
Program
Program* Log Output Data (3)
REGS_MATCHED
Filter and Sort Query Builder Data Describe Graph Analyze Export Send To
Ready
sasmeta

	id	cstat	county	mstat
1	1	z		a
2	2	y	02	b
3	2	x	18	b
4	3	w	03	c
5	3	w	04	d
6	4	v	05	e
7	4	u	06	f
8	4	u	07	g
9	5		08	h
10	6	t	14	
11	7		09	i
12	7		10	j
13	8	s	12	
14	8	r	11	

As an observation is only read once, the last read will come from the last-mentioned dataset for all but the second and above observations in a many:1 match. For those the last read will come from the first mentioned dataset. In a 1:many match the last read will always come from the last mentioned dataset.

The default match with the *Merge* statement is similar to a full join in the way that all observations that do not match will be added to the output dataset. If we want to have matches similar to other joins we will have to add some options and statements to our program. For each of the datasets we can create a temporary variable which is true (1) when the dataset contributes to the match and false (0) when it doesn't. Then we can use these variables to decide what output we want to put on the result dataset.

Here is an example of a program where only the observations that match will be written to the output dataset:

```
data regs_matched ;
   merge reg1s (in=ds1)
         reg2s (in=ds2)
         ;
   by id ;
   if ds1 and ds2 then
    output;
run;
```

For a left match we change the condition:

```
   if ds1 then
    output;
```

A right match can have this condition:

```
   if ds2 then
    output;
```

Another way to match files with Sas is to use the *Update* statement instead of the *Merge* statement. The *Update* statement will use the first mentioned file as a master table and the second as a transaction table. The algorithm for *Update* is different form *Merge*. With *Update* the first observation in the master dataset with matching key will be updated with the observations from the transaction dataset. If there are more than one observation with the same key in the transaction dataset, the value from the last one will be the updated one. When it comes to missing values, the default is that missing values will not override values in the master dataset. We can force missing values to override by using the option updatemode=nomissingcheck in the *Update* statement:

```
data reg_updated_opp_miss ;
   update reg2s
          reg1s updatemode=nomissingcheck
         ;
   by id ;
run;
```

14.2. Spss

We create the two datasets with this syntax in Spss:

```
DATA LIST/
  id      1-1 (A)
  mstat   3-3 (A)
  county  5-6 (A)
  .
BEGIN DATA
3 c 03
1 a
2 b 02
4 e 05
3 d 04
4 f 06
4 g 07
7 i 09
5 h 08
7 j 10
END DATA.
SORT CASES BY id.
SAVE /OUTFILE='reg1.sav'.
DATA LIST/
  id      1-1 (A)
  cstat   3-3 (A)
  county  5-6 (A)
  .
BEGIN DATA
2 y 19
1 z 20
2 x 18
4 v 16
3 w 17
4 u 15
8 s 12
6 t 14
8 r 11
END DATA.
SORT CASES BY id.
SAVE /OUTFILE='reg2.sav'.
```

The datasets we shall match have to be sorted on the key variables before we match. We can match files with two different commands: *Match files* and *Update files*. The *Match files* command is similar to the *Merge* statement in Sas, however the algorithms are different. We have two different ways to use the *Match files* command; one is to match two files where both provide observations to the new dataset, the second is to use one of the datasets as a lookup table. Observations from the lookup table with keys that is not found in the other dataset will not be added to the output dataset. A lookup dataset is defined with the *Table* subcommand and may not have duplicates. Here is syntax for matching the two files:

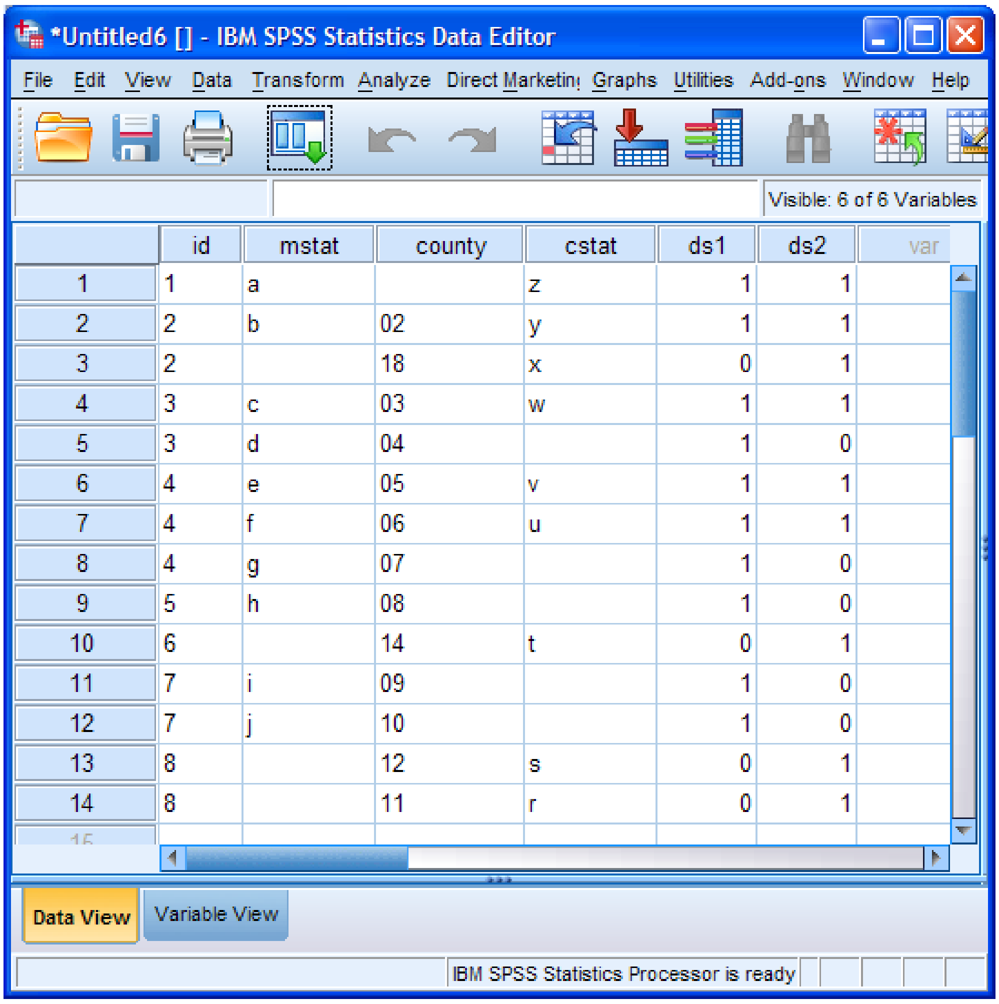
```
MATCH FILES FILE='reg1.sav'
           /IN=ds1
           /FILE='reg2.sav'
           /IN=ds2
           /BY id
           .

EXECUTE.
```

The *File* subcommands name the files to match. With the *In* subcommand, we create a variable which shows when the file contributes to the match. This is used in a similar way to the *In* option in Sas. The key variables are defined with the *By* subcommand.

In an ordinary match Spss will always match 1:1 even if there are duplicates in the datasets. If there are duplicates a warning is written to the log:

```
Warning # 5132

Duplicate key in a file.  The BY variables do not uniquely identify each case

on the indicated file.  Please check the results carefully.
```

The output from the match is shown here:


*Untitled6 [] - IBM SPSS Statistics Data Editor
File Edit View Data Transform Analyze Direct Marketing Graphs Utilities Add-ons Window Help
Visible: 6 of 6 Variables
Data View
Variable View
IBM SPSS Statistics Processor is ready

	id	mstat	county	cstat	ds1	ds2
1	1	a		z	1	1
2	2	b	02	y	1	1
3	2		18	x	0	1
4	3	c	03	w	1	1
5	3	d	04		1	0
6	4	e	05	v	1	1
7	4	f	06	u	1	1
8	4	g	07		1	0
9	5	h	08		1	0
10	6		14	t	0	1
11	7	i	09		1	0
12	7	j	10		1	0
13	8		12	s	0	1
14	8		11	r	0	1

As it is only 1:1 matches that is used, we see that when there are duplicates, they only match as long as matches are found by matching 1:1. This is illustrated by looking at the values of the variables *ds1* and *ds2*. These variables are not needed for the match, but they are useful when we want to select observations based on the result of the match (as seen below). Uneven observations will not be matched, but they will be added to the output data. We see that for observations with ids 2, 3 and 4. Another difference from the Sas merge is that when there are common variables which are not key variables, the value from the first dataset is kept when they match. If they don't match the value from the contributing dataset will be kept.

We can use the *In* variables to choose which observations to keep on the output dataset. We use the *Select If* command after the match to choose which observations to keep. The syntax for keeping the observations which match:

```
SELECT IF(ds1 AND ds2).
EXECUTE.
```

If we want all the observations from the first (left) dataset to the output, we use this instead:

```
SELECT IF(ds1).
EXECUTE.
```

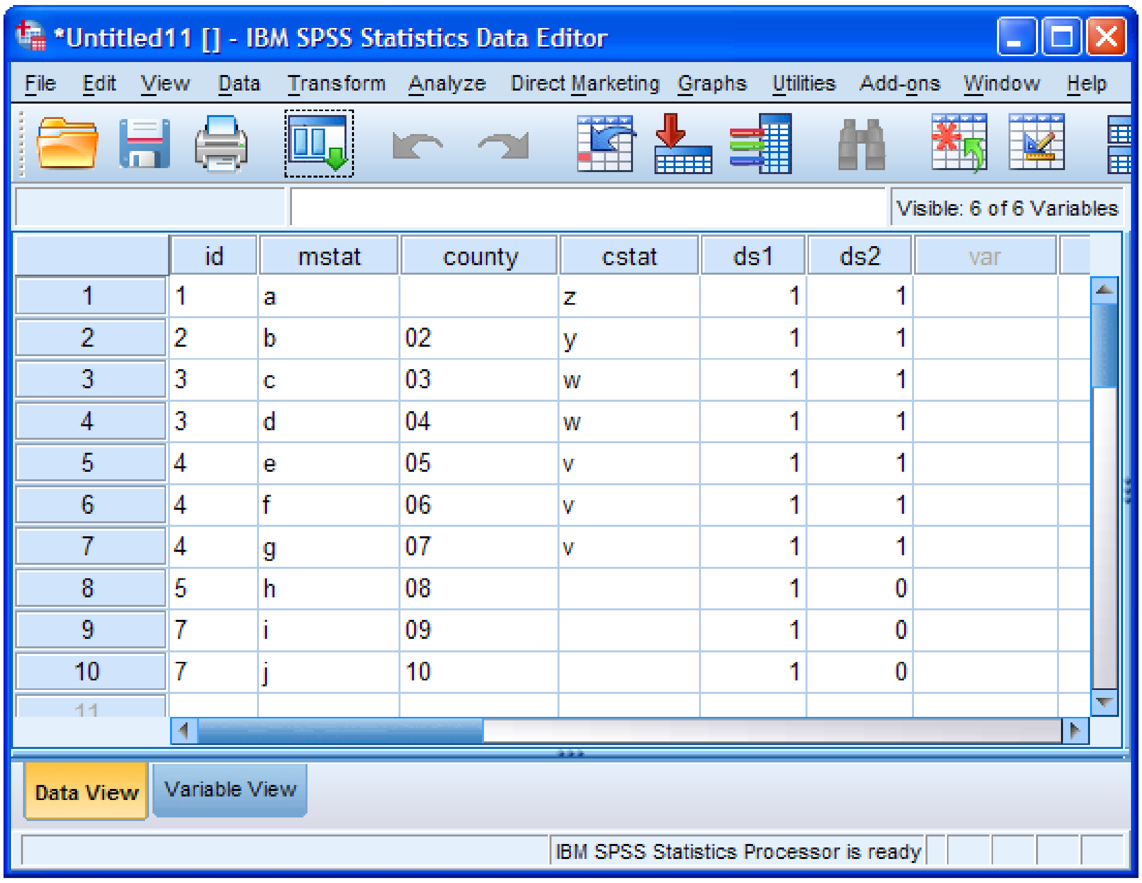
The other way round, observations from the last (right) dataset kept:

```
SELECT IF(ds2).
EXECUTE.
```

The match with using a lookup table demands that the lookup table is without duplicates. We can delete the duplicates in reg2s and then match the files by using the reg2s without duplicates as a lookup table:

```
MATCH FILES FILE='reg2.sav'
            /FIRST=first_id
            /BY id
            .
SELECT IF (first_id).
EXECUTE.

DELETE VARIABLES first_id.

MATCH FILES FILE='reg1.sav'
            /IN=ds1
            /TABLE=*
            /IN=ds2
            /BY id
            .
EXECUTE.
```

We use the *Table* subcommand and name the file with *. This means we will use the active dataset as input. The active dataset is the *reg2s* without duplicates. Now there will be many:1 matches as well as 1:1 matches. When the datasets don't match all observations from the first dataset will be kept, but none from the lookup dataset:


*Untitled11 [] - IBM SPSS Statistics Data Editor
File Edit View Data Transform Analyze Direct Marketing Graphs Utilities Add-ons Window Help
Visible: 6 of 6 Variables
Data View
Variable View
IBM SPSS Statistics Processor is ready

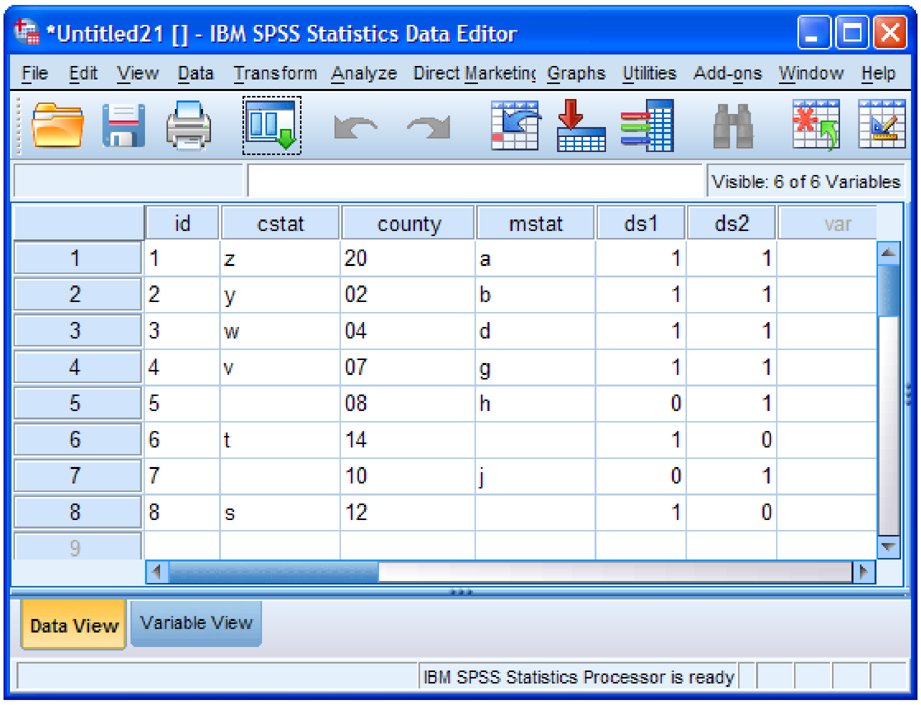
	id	mstat	county	cstat	ds1	ds2
1	1	a		z	1	1
2	2	b	02	y	1	1
3	3	c	03	w	1	1
4	3	d	04	w	1	1
5	4	e	05	v	1	1
6	4	f	06	v	1	1
7	4	g	07	v	1	1
8	5	h	08		1	0
9	7	i	09		1	0
10	7	j	10		1	0

The order of the datasets is important here as well when there are non-key variables with same names in both datasets, therefore it should be avoided.

When we want to have an update instead of a match, the master dataset is not allowed to have duplicates. If there are duplicates on the transaction dataset the value from the last observation within the group of observations with the same key will be picked. Here is a syntax where we first delete duplicates on the master dataset and then use the *Update* command:

```
MATCH FILES FILE='reg2.sav'
           /FIRST=first_id
           /BY id
           .
SELECT IF (first_id).
EXECUTE.

DELETE VARIABLES first_id.

UPDATE FILE=*
      /IN=ds1
      /FILE='reg1.sav'
      /IN=ds2
      /BY id
      .
EXECUTE.
```

The output dataset will now be like this:


*Untitled21 [] - IBM SPSS Statistics Data Editor
File Edit View Data Transform Analyze Direct Marketing Graphs Utilities Add-ons Window Help
Visible: 6 of 6 Variables
Data View
Variable View
IBM SPSS Statistics Processor is ready

	id	cstat	county	mstat	ds1	ds2	var
1	1	z	20	a	1	1	
2	2	y	02	b	1	1	
3	3	w	04	d	1	1	
4	4	v	07	g	1	1	
5	5		08	h	0	1	
6	6	t	14		1	0	
7	7		10	j	0	1	
8	8	s	12		1	0	
9							

If a variable value from the transaction dataset is missing it will not replace the value in the master dataset.

14.3. Stata

We use the *joinby* or *merge* command to match datasets in Stata, but the datasets must be sorted on the key variables before the matching. Here is the syntax for creating the two datasets:

```
clear
input str1 id str1 mstat str2 county
3 c 03
1 a
2 b 02
4 e 05
3 d 04
4 f 06
4 g 07
7 i 09
5 h 08
7 j 10
end
sort id, stable
save "reg1.dta", replace
clear
input str1 id str1 cstat str2 county
2 y 19
1 z 20
2 x 18
4 v 16
3 w 17
4 u 15
8 s 12
6 t 14
8 r 11
end
sort id, stable
```

```
save "reg2.dta", replace
```

We will now look at the *joinby* command, which joins datasets in a similar way to a sql join. The syntax may look like this for an inner join:

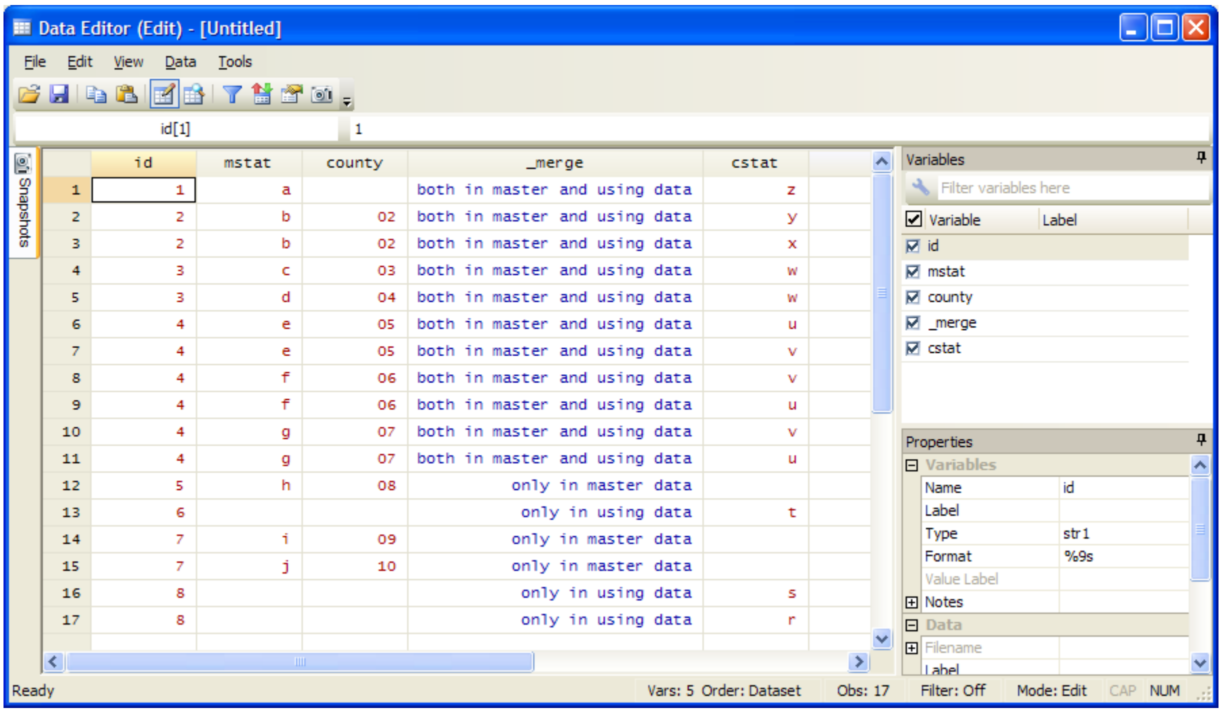
```
clear
use "reg1.dta"
joinby id using "reg2.dta"
```

The matching algorithm is the same as in *Proc sql* in Sas; all observations with a key in the first dataset match all observations with the same key in the second dataset. The difference between *Proc sql* in Sas and *joinby* is when there are common non-key variables. In *Proc sql* the common variables will usually be separated in two variables in the output dataset. With the *joinby* command we will have only one variable in the output dataset, but the content may be picked from both datasets. When they match the values are taken from the first dataset mentioned unless we use the update and replace options. If so, non-missing vales are taken from the second dataset, otherwise values from the first dataset are used. If they don't match and we choose to add the observation to the output, the values will be taken from the dataset mentioned in the *use* command.

The default match with the *joinby* command is similar to an inner join. If we want a full join, we have to add the *unmatched* option:

```
clear
use "reg1.dta"
joinby id using "reg2.dta", unmatched(both)
```

The output data will be like this; an extra variable, _merge, is added:



	id	mstat	county	_merge	cstat
1	1	a		both in master and using data	z
2	2	b	02	both in master and using data	y
3	2	b	02	both in master and using data	x
4	3	c	03	both in master and using data	w
5	3	d	04	both in master and using data	w
6	4	e	05	both in master and using data	u
7	4	e	05	both in master and using data	v
8	4	f	06	both in master and using data	v
9	4	f	06	both in master and using data	u
10	4	g	07	both in master and using data	v
11	4	g	07	both in master and using data	u
12	5	h	08	only in master data	
13	6			only in using data	t
14	7	i	09	only in master data	
15	7	j	10	only in master data	
16	8			only in using data	s
17	8			only in using data	r

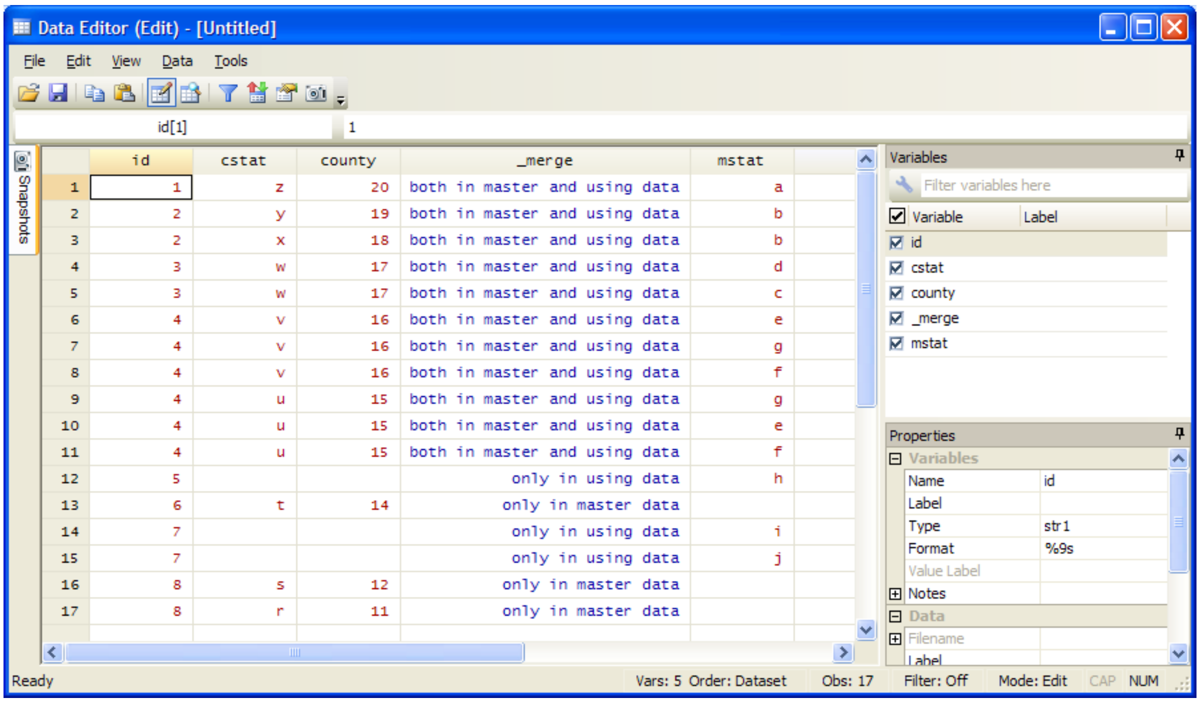
Name	id
Label	
Type	str1
Format	%9s
Value Label	
Notes	
Data	
Filename	
Label	

Ready Vars: 5 Order: Dataset Obs: 17 Filter: Off Mode: Edit CAP NUM

If we change the order of the files, the commands will be like this:

```
clear
use "reg2.dta"
joinby id using "reg1.dta", unmatched(both)
```

We see that the content of the common variable *county* has changed and also the order of the variables:



Data Editor (Edit) - [Untitled]

id[1] 1

	id	cstat	county	_merge	mstat
1	1	z	20	both in master and using data	a
2	2	y	19	both in master and using data	b
3	2	x	18	both in master and using data	b
4	3	w	17	both in master and using data	d
5	3	w	17	both in master and using data	c
6	4	v	16	both in master and using data	e
7	4	v	16	both in master and using data	g
8	4	v	16	both in master and using data	f
9	4	u	15	both in master and using data	g
10	4	u	15	both in master and using data	e
11	4	u	15	both in master and using data	f
12	5			only in using data	h
13	6	t	14	only in master data	
14	7			only in using data	i
15	7			only in using data	j
16	8	s	12	only in master data	
17	8	r	11	only in master data	

Variables: id, cstat, county, _merge, mstat

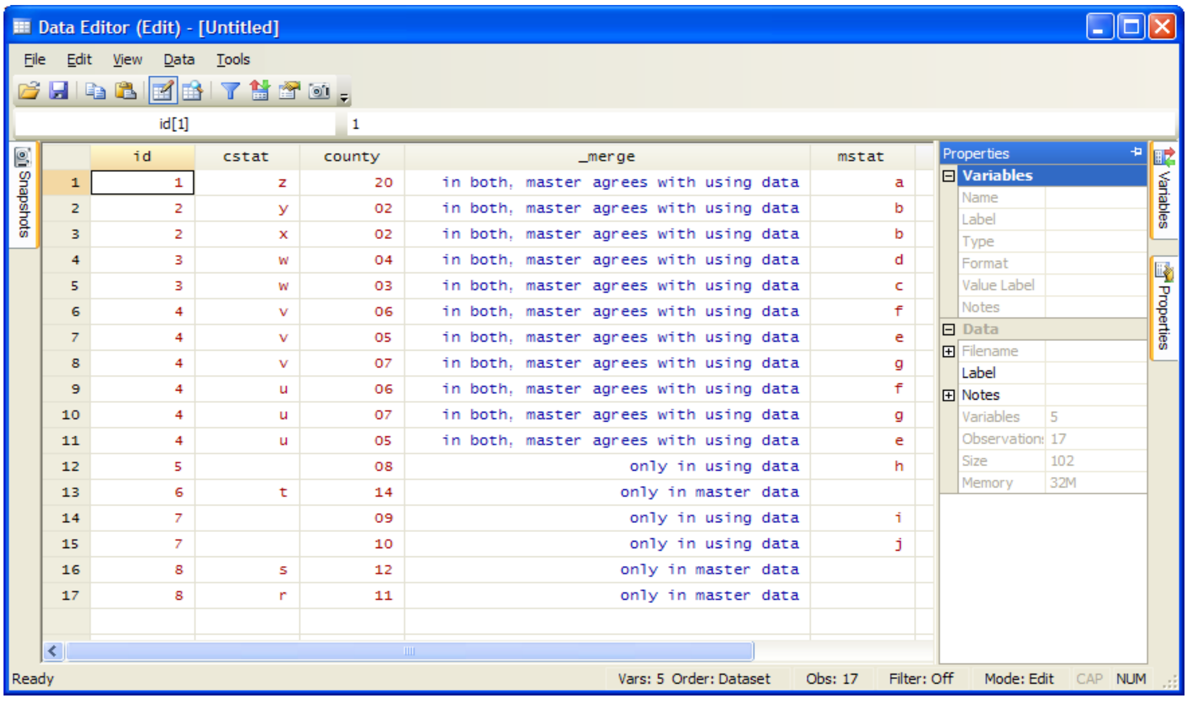
Properties: Name id; Type str1; Format %9s

Ready Vars: 5 Order: Dataset Obs: 17 Filter: Off Mode: Edit CAP NUM

The content of the common variables changes for non-missing values when we add the *update* and *replace* options:

```
use "reg2.dta", clear
joinby id using "reg1.dta", unmatched(both) update replace
```

This is like the previous *joinby* example, except where id equals 1, 6 and 8 where values from the first dataset are kept because they are missing in the second. The values of the _merge variable have also changed when the datasets match:



Data Editor (Edit) - [Untitled]

id[1] 1

	id	cstat	county	_merge	mstat
1	1	z	20	in both, master agrees with using data	a
2	2	y	02	in both, master agrees with using data	b
3	2	x	02	in both, master agrees with using data	b
4	3	w	04	in both, master agrees with using data	d
5	3	w	03	in both, master agrees with using data	c
6	4	v	06	in both, master agrees with using data	f
7	4	v	05	in both, master agrees with using data	e
8	4	v	07	in both, master agrees with using data	g
9	4	u	06	in both, master agrees with using data	f
10	4	u	07	in both, master agrees with using data	g
11	4	u	05	in both, master agrees with using data	e
12	5		08	only in using data	h
13	6	t	14	only in master data	
14	7		09	only in using data	i
15	7		10	only in using data	j
16	8	s	12	only in master data	
17	8	r	11	only in master data	

Properties: Variables 5; Observations 17; Size 102; Memory 32M

Ready Vars: 5 Order: Dataset Obs: 17 Filter: Off Mode: Edit CAP NUM

If we want a left join, we can use these commands:

```
clear
use "reg1.dta"
joinby id using "reg2.dta", unmatched(master)
```

A right join is executed with this syntax:

```
clear
use "reg1.dta"
joinby id using "reg2.dta", unmatched(using)
```

Another way to merge the files is by using the *merge* command. Stata strongly recommends not matching files with duplicates on both files with the *merge* command. Nevertheless, it is interesting to know the algorithm used. Here is syntax for a match with the *merge* command:

```
clear
use "reg1.dta"
merge id using "reg2.dta"
sort id, stable
```

When we use the *merge* command without telling what kind of match we want, we get some notes in the viewer:

```
(note: you are using old merge syntax; see [D] merge for new syntax)

variable id does not uniquely identify observations in the master data

variable id does not uniquely identify observations in reg2.dta
```

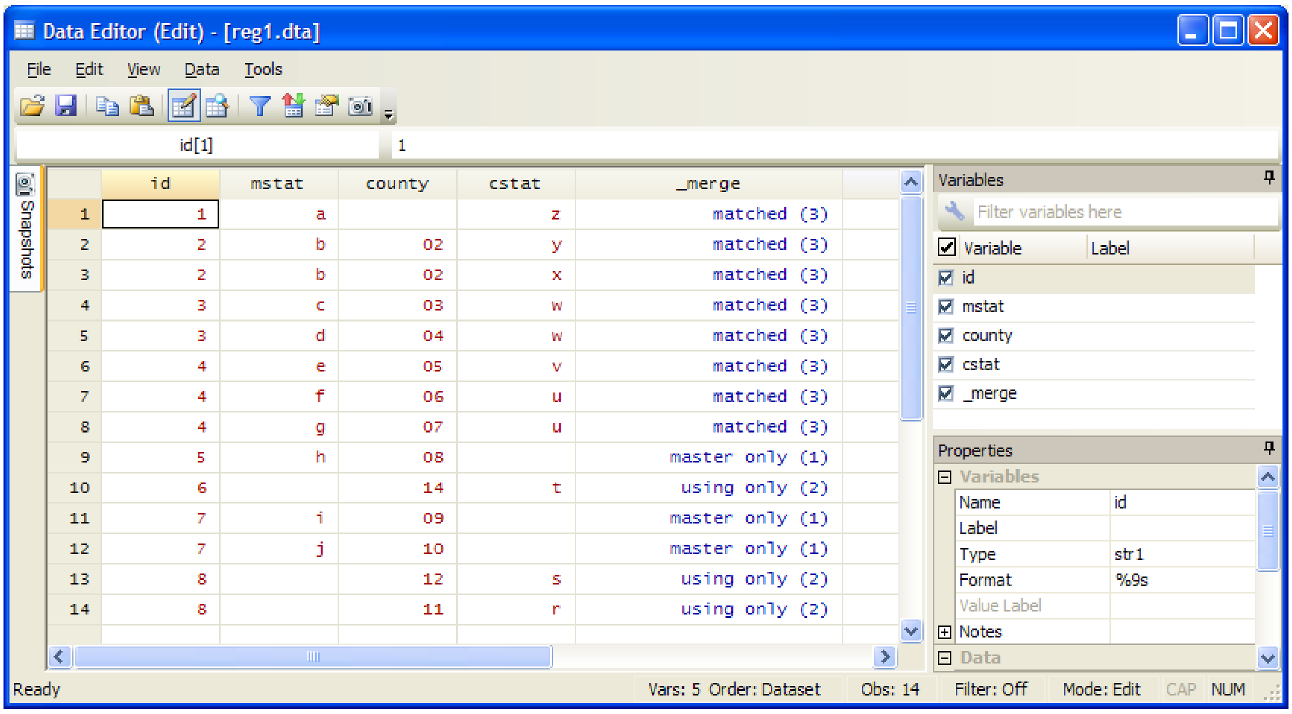
The result will be the same if we add the match type (see page 74) to the command, but the notes disappear:

```
clear
use "reg1.dta"
merge m:m id using "reg2.dta"
```

Instead, we get other information:

```
    Result                           # of obs.

    -----------------------------------------

    not matched                             6

        from master                         3  (_merge==1)

        from using                          3  (_merge==2)

    matched                                 8  (_merge==3)

    -----------------------------------------
```

The output dataset from the matching is this:


Data Editor (Edit) - [reg1.dta]

id[1] 1

	id	mstat	county	cstat	_merge
1	1	a		z	matched (3)
2	2	b	02	y	matched (3)
3	2	b	02	x	matched (3)
4	3	c	03	w	matched (3)
5	3	d	04	w	matched (3)
6	4	e	05	v	matched (3)
7	4	f	06	u	matched (3)
8	4	g	07	u	matched (3)
9	5	h	08		master only (1)
10	6		14	t	using only (2)
11	7	i	09		master only (1)
12	7	j	10		master only (1)
13	8		12	s	using only (2)
14	8		11	r	using only (2)

Variable	Label
id	
mstat	
county	
cstat	
_merge	

Properties	
Name	id
Label	
Type	str1
Format	%9s
Value Label	
Notes	

Ready Vars: 5 Order: Dataset Obs: 14 Filter: Off Mode: Edit CAP NUM

The algorithm is like the *Merge* in Sas, except for the common non-key variable *county*. Its values are taken from the first dataset as long as they contribute to the match. This can be changed by using the update and replace options:

```
use "reg1.dta", clear
merge m:m id using "reg2.dta", update replace
```

A new variable, *_merge*, is also added to the output dataset. This variable tells which observations that match and from what dataset the non-matching observations are taken. The order of the datasets is important in Stata as well when it comes to these common non-key variables as their values will be different if the order changes.

To keep only the observations which match, we use the keep option like this:

```
use "reg1.dta", clear
merge m:m id using "reg2.dta", keep(matched)
```

For keeping all observations from the first dataset we use this command:

```
use "reg1.dta", clear
merge m:m id using "reg2.dta", keep(matched master)
```

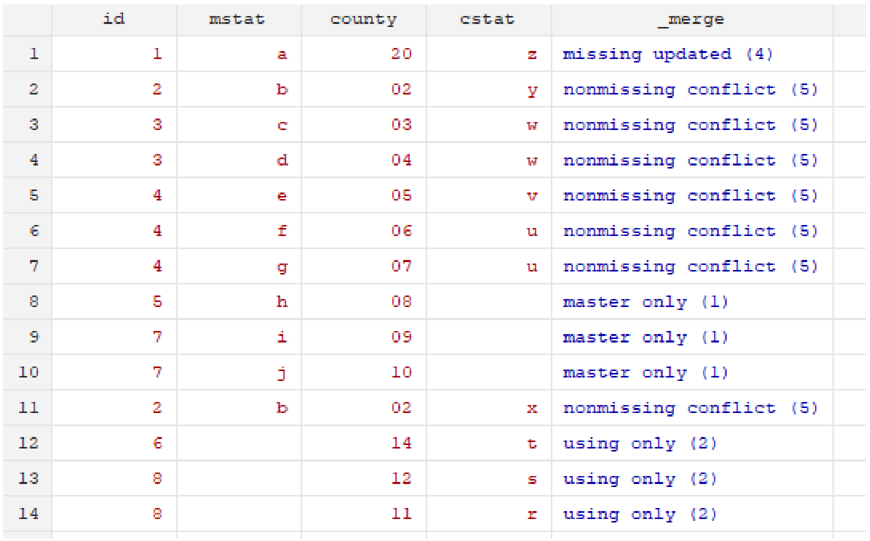
The other way around:

```
use "reg1.dta", clear
merge m:m id using "reg2.dta", keep(matched using)
```

We can add an *update* option to the merge, then it will use the second dataset for updating the common variables (here: *county*). The syntax is like this:

```
merge m:m id using "reg2.dta", update
```

The result shows that only missing values from the first dataset will be updated. If the first dataset already has a valid value for the variable, it will keep that value. The variable *_merge* tells us how the merge went:



	id	mstat	county	cstat	_merge
1	1	a	20	z	missing updated (4)
2	2	b	02	y	nonmissing conflict (5)
3	3	c	03	w	nonmissing conflict (5)
4	3	d	04	w	nonmissing conflict (5)
5	4	e	05	v	nonmissing conflict (5)
6	4	f	06	u	nonmissing conflict (5)
7	4	g	07	u	nonmissing conflict (5)
8	5	h	08		master only (1)
9	7	i	09		master only (1)
10	7	j	10		master only (1)
11	2	b	02	x	nonmissing conflict (5)
12	6		14	t	using only (2)
13	8		12	s	using only (2)
14	8		11	r	using only (2)

The difference from the merge without the update option is that missing values in the first dataset will be updated from the second, which is the case for the first observation in the result. We see that in the *_merge* variable. When the first dataset already has a value, the *_merge* will contain a message of a nonmissing conflict.

14.4.R

In R the matching algorithm follows the join as in Sas *proc sql* and Stata *joinby*. We can create the same datasets in R:

```
id <-c("3","1","2","4","3","4","4","7","5","7")
mstat <-c("c","a","b","e","d","f","g","i","h","j")
county <- c("03","","02","05","04","06","07","09","08","10")
reg1 <- data.frame(id,mstat,county)
print(reg1)
id<-c("2","1","2","4","3","4","8","6","8")
cstat<-c("y","z","x","v","w","u","s","t","r")
county<-c("19","20","18","06","17","15","12","14","11")
reg2 <- data.frame(id,cstat,county)
print(reg2)
```

Now we can match with a full join. We use the *dplyr* package with its joins for this and use the *arrange* command to sort the output data:

```
fulljoin <- full_join(x=reg1,y=reg2,by="id") %>% arrange(id)
print(fulljoin)
```

The result:

```
   id mstat county.x cstat county.y

1   1     a                z       20
2   2     b         02     y       19
3   2     b         02     x       18
4   3     c         03     w       17
5   3     d         04     w       17
6   4     e         05     v       06
7   4     e         05     u       15
8   4     f         06     v       06
```

```
9    4      f        06      u        15
10   4      g        07      v        06
11   4      g        07      u        15
12   5      h        08   <NA>      <NA>
13   6   <NA>      <NA>      t        14
14   7      i        09   <NA>      <NA>
15   7      j        10   <NA>      <NA>
16   8   <NA>      <NA>      s        12
17   8   <NA>      <NA>      r        11
```

All combinations of rows with the same value of id have been added to the output. The common variable county which was not part of the match key has been added separately from each data frame. The names have a suffix, .x for the variable from the data frame mentioned in the *x* argument and .y for the data frame mentioned in *y* argument.

To do an inner join we replace *full_join* with *inner_join*:

```
innerjoin <- inner_join(x=reg1,y=reg2,by="id") %>% arrange(id)
print(innerjoin)
```

Now only those who match are added to the output data frame:

```
   id mstat county.x cstat county.y

1    1      a              z        20
2    2      b        02    y        19
3    2      b        02    x        18
4    3      c        03    w        17
5    3      d        04    w        17
6    4      e        05    v        06
7    4      e        05    u        15
8    4      f        06    v        06
9    4      f        06    u        15
10   4      g        07    v        06
11   4      g        07    u        15
```

For left join and right join we change the name of joins:

```
leftjoin <- left_join(x=reg1,y=reg2,by="id") %>% arrange(id)
print(leftjoin)
rightjoin <- right_join(reg1,reg2,by="id") %>% arrange(id)
print(rightjoin)
```

We may also find rows in the first data frame that has one or more matches in the second data frame with a *semi_join*. This will not join the files, only output those from the first data frame that has a match in the second:

```
semijoin <- semi_join(reg1,reg2,by="id") %>% arrange(id)
print(semijoin)
```

These rows have a match:

```
  id mstat county

1   1      a
2   2      b      02
3   3      c      03
4   3      d      04
5   4      e      05
6   4      f      06
7   4      g      07
```

To do the opposite, find those who don't match, we can use *anti_join*:

```
antijoin1 <- anti_join(reg1,reg2,by="id") %>% arrange(id)
print(antijoin1)
```

Now we see those who don't match:

```
  id mstat county

1  5     h     08
2  7     i     09
3  7     j     10
```

Instead of using joins in *dplyr* we may use the merge command. It will match the files as the joins, but the syntax is different. Here we have examples on full, inner, left and right joins:

```
reg1and2fj<-merge(x = reg1, y = reg2, by = "id", all = TRUE)
print(reg1and2fj)
reg1and2ij<-merge(x = reg1, y = reg2, by = "id", all = FALSE)
print(reg1and2ij)
reg1and2lj<-merge(x = reg1, y = reg2, by = "id", all.x = TRUE)
print(reg1and2lj)
reg1and2rj<-merge(x = reg1, y = reg2, by = "id", all.y = TRUE)
print(reg1and2rj)
```

We define the type of join arguments all, all.x and all.y

• Full join	all=TRUE or all.x=TRUE and all.y=TRUE
• Inner join	all=FALSE
• Left join	all.x=TRUE
• Right join	all.y=TRUE

There is a possibility to update values from one data frame with values from another data frame. However, there are some restrictions with this update: key variables must be unique, so we must delete duplicates on both files. Furthermore, key values in the second data frame must exist in the first data frame. That means we must delete rows in the second data frame that does not have the same key in the first data frame before we update. Finally, it is not allowed to have variables in the second data frame that is not present in the first. Here is some code that prepares the data frames for update:

```
reg1u <- reg1[!duplicated(reg1[c(1)]),]
reg2x <- subset(reg2, select = -cstat)
reg2u <- reg2x[!duplicated(reg2x[c(1)]),]
reg2u <- semi_join(reg2u,reg1,by="id") %>% arrange(id)
```

Now we can update the first dataset with values from the second with *rows_update* in *dplyr*:

```
reg1u %>%
  rows_update(reg2u, by = "id")
```

The result shows that rows where the keys are found in the first dataset are updated with data from second data frame:

```
  id mstat county

1  3     c     17
2  1     a     20
3  2     b     19
4  4     e     06
8  7     i     09
9  5     h     08
```

14.5. Python

The matching in Python is done with the merge command. The syntax is similar to the merge command in R and the match algorithm is identical to join as in Sas proc sql, Stata joinby and merge in R.

We start with creating two files that contains all the possible match situations:

```
reg1file="""
3 c 03
1 a
2 b 02
4 e 05
3 d 04
4 f 06
4 g 07
7 i 09
5 h 08
7 j 10
"""
reg1 = pd.read_csv(
        StringIO(reg1file),
        names=['id', 'mstat', 'county'],
        dtype=object,
        header=None,
        sep=' '
)

reg2file="""
2 y 19
1 z 20
2 x 18
4 v 06
3 w 17
4 u 15
8 s 12
6 t 14
8 r 11
"""
reg2 = pd.read_csv(
        StringIO(reg2file),
        names=['id', 'cstat', 'county'],
        dtype=object,
        header=None,
        sep=' '
)
display(reg1, reg2)
```

The data frames are here:

	id	mstat	county
0	3	c	03
1	1	a	NaN
2	2	b	02
3	4	e	05
4	3	d	04
5	4	f	06
6	4	g	07
7	7	i	09
8	5	h	08
9	7	j	10

	id	cstat	county
0	2	y	19
1	1	z	20
2	2	x	18
3	4	v	06
4	3	w	17
5	4	u	15
6	8	s	12
7	6	t	14
8	8	r	11

We start with the easiest merge where all with the same ids will match and only those who match will be included in the output. This is called an inner join:

```
pd.merge(reg1, reg2, on='id')
```

We see that the variable county, which is not a key, but are in both datasets, will be included from both data frames (with new names):

	id	mstat	county_x	cstat	county_y
0	3	c	03	w	17
1	3	d	04	w	17
2	1	a	NaN	z	20
3	2	b	02	y	19
4	2	b	02	x	18
5	4	e	05	v	06
6	4	e	05	u	15
7	4	f	06	v	06
8	4	f	06	u	15
9	4	g	07	v	06
10	4	g	07	u	15

If we want to sort the result by the id, we can add the *sort_values* function:

```
pd.merge(reg1, reg2, on='id').sort_values('id')
```

Now the result is sorted:

	id	mstat	county_x	cstat	county_y
2	1	a	NaN	z	20
3	2	b	02	y	19
4	2	b	02	x	18
0	3	c	03	w	17
1	3	d	04	w	17
5	4	e	05	v	06
6	4	e	05	u	15
7	4	f	06	v	06
8	4	f	06	u	15
9	4	g	07	v	06
10	4	g	07	u	15

We can specify the key for each data frame. This is useful when the keys have different names:

```
pd.merge(reg1, reg2, left_on='id', right_on='id').sort_values('id')
```

We can also define the suffixes for common variables. This example will change the suffixes from _x and _y to _1 and _2:

```
pd.merge(reg1, reg2, on='id', suffixes=('_1', '_2')).sort_values('id')
```

Now we can merge with a full join, all with the same id matches and all id's that don't match will be included in the output. We add an indicator as well to show how the match went:

```
pd.merge(reg1, reg2, on='id', how='outer', indicator=True).sort_values('id')
```

Now we see how the data frames matched:

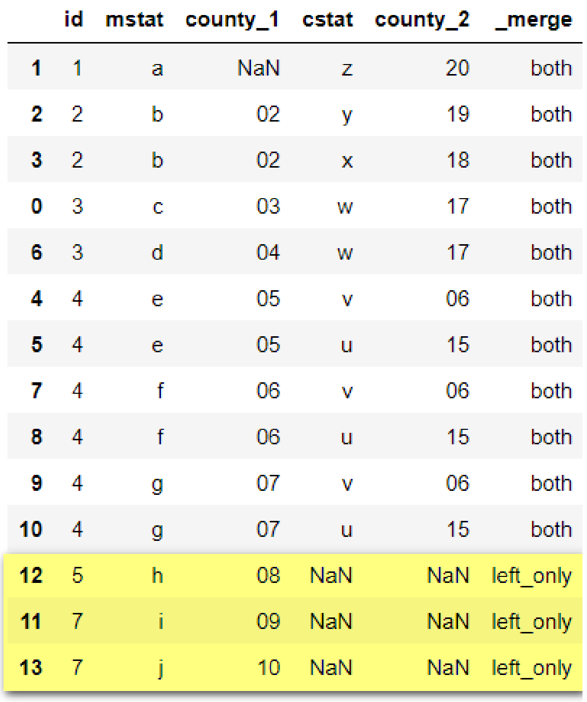
	id	mstat	county_x	cstat	county_y	_merge
2	1	a	NaN	z	20	both
3	2	b	02	y	19	both
4	2	b	02	x	18	both
0	3	c	03	w	17	both
1	3	d	04	w	17	both
10	4	g	07	u	15	both
9	4	g	07	v	06	both
8	4	f	06	u	15	both
6	4	e	05	u	15	both
5	4	e	05	v	06	both
7	4	f	06	v	06	both
13	5	h	08	NaN	NaN	left_only
16	6	NaN	NaN	t	14	right_only
11	7	i	09	NaN	NaN	left_only
12	7	j	10	NaN	NaN	left_only
15	8	NaN	NaN	r	11	right_only
14	8	NaN	NaN	s	12	right_only

Here we have examples on left and right joins:

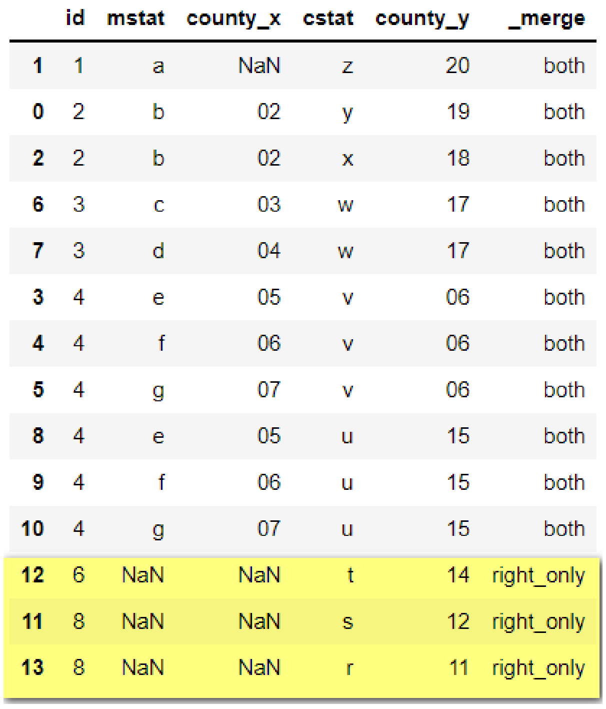
```
pd.merge(reg1, reg2, on='id', how='left', indicator=True).sort_values('id')
```

```
pd.merge(reg1, reg2, on='id', how='right', indicator=True).sort_values('id')
```

We see that in addition to the rows that match, the rows that are only in the left dataset will be added to the output in the left merge. Opposite, when we use the right merge rows that are only in the right dataset will be added to the output:



	id	mstat	county_1	cstat	county_2	_merge
1	1	a	NaN	z	20	both
2	2	b	02	y	19	both
3	2	b	02	x	18	both
0	3	c	03	w	17	both
6	3	d	04	w	17	both
4	4	e	05	v	06	both
5	4	e	05	u	15	both
7	4	f	06	v	06	both
8	4	f	06	u	15	both
9	4	g	07	v	06	both
10	4	g	07	u	15	both
12	5	h	08	NaN	NaN	left_only
11	7	i	09	NaN	NaN	left_only
13	7	j	10	NaN	NaN	left_only



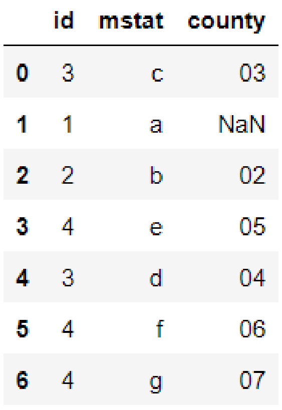
	id	mstat	county_x	cstat	county_y	_merge
1	1	a	NaN	z	20	both
0	2	b	02	y	19	both
2	2	b	02	x	18	both
6	3	c	03	w	17	both
7	3	d	04	w	17	both
3	4	e	05	v	06	both
4	4	f	06	v	06	both
5	4	g	07	v	06	both
8	4	e	05	u	15	both
9	4	f	06	u	15	both
10	4	g	07	u	15	both
12	6	NaN	NaN	t	14	right_only
11	8	NaN	NaN	s	12	right_only
13	8	NaN	NaN	r	11	right_only

All id values from both data frames match when there are duplicates. That is the same in all these merges.

If we just want to see which id's in reg1 that is also in reg2 without actually match the two data frames, we can do a lookup (or a semi-join) using the *isin* method and check when it is true:

```
reg1.loc[reg1['id'].isin(reg2['id']) == True]
```

These rows from reg1 are also found with the same id in reg2:



	id	mstat	county
0	3	c	03
1	1	a	NaN
2	2	b	02
3	4	e	05
4	3	d	04
5	4	f	06
6	4	g	07

We can find those in reg1 who are not found in reg2 the same way, except that we select those who have *isin* false. This will be a so-called anti-join:

```
reg1.loc[reg1['id'].isin(reg2['id']) == False]
```

These rows from reg1 does not match any id's in reg2:

	id	mstat	county
7	7	i	09
8	5	h	08
9	7	j	10

When we have more than one key, we can use conditions like these:

```
reg1.loc[(reg1['id'].isin(reg2['id']) == True) &
(reg1['county'].isin(reg2['county']) == True)]
```

It gives us the single row that are in both data frames:

	id	mstat	county
5	4	f	06

Another way to merge files is to use one file to update another. We can do that with the update method. This method uses the row index as matching key; hence we have to set our identification variables to an index. It does not allow duplicate indexes, so we must delete duplicates on both files before we do the update. Here is a syntax to drop duplicates and make the *id* column an index for both files. Then we update reg1i with reg2i. Finally, we reset the index so that the *id* column will be an ordinary column again

```
reg1u = reg1[~reg1.duplicated(['id'], keep='first')].sort_values(['id'])
reg1i = reg1u.set_index('id')
reg2u = reg2[~reg2.duplicated(['id'], keep='first')].sort_values(['id'])
reg2i = reg2u.set_index('id')
reg1i.update(reg2i)
reg1i = reg1i.reset_index()
reg1i
```

Rows with *id* values that is not in the first data frame will not be added to the result data frame, nor will columns that are only in the second file, as we can see from the result:

	id	mstat	county
0	1	a	20
1	2	b	19
2	3	c	17
3	4	e	06
4	5	h	08
5	7	i	09

14.6. A matching comparison

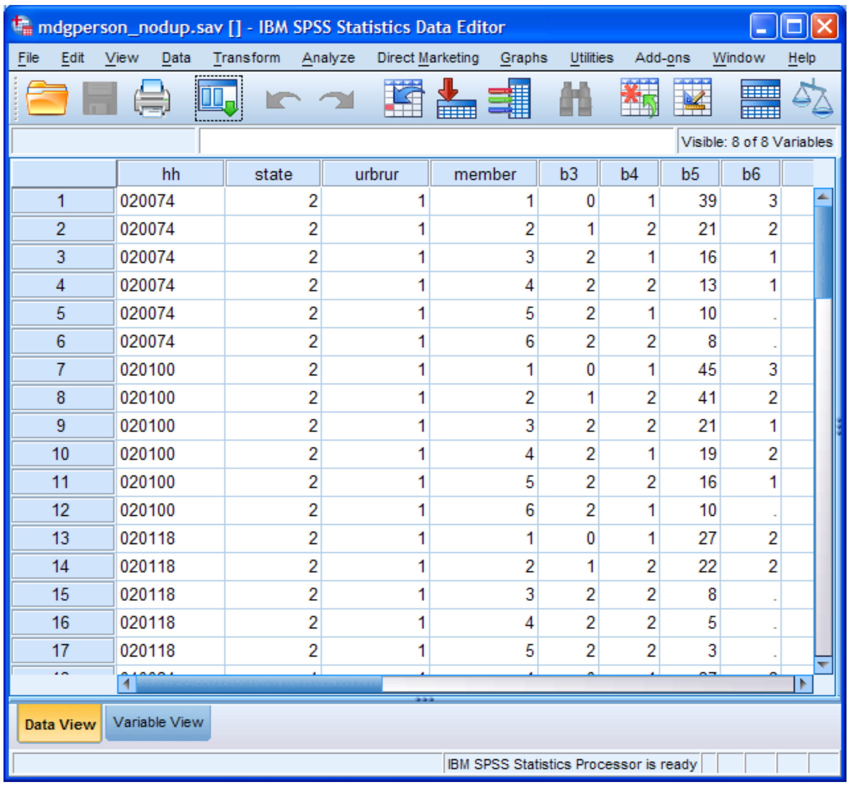
Here is a summation of matching in Sas, Spss, Stata, R and Python:

Match type	**Sas**		**Spss**		**Stata**		**R**		**Python**
	Proc sql join	**Merge**	**Match files with File**	**Match files with Table**	**joinby**	**merge**	**dplyr join**	**merge**	**pd.merge**
1:1	1:1 match	1:1 match	1:1 match	1:1 match	1:1 match	1:1 match	1:1 match	1:1 match	1:1 match
1:m	1 matches all	1 matches all	Divided into 1:1 and 0:m	Not allowed	1 matches all	1 matches all	1 matches all	1 matches all	1 matches all
m:1	All matches 1	All matches 1	Divided into 1:1 and m:0	All matches 1	All matches 1	All matches 1	All matches 1	All matches 1	All matches 1
m:m	All matches all	Divided into 1:1, 1:1 etc. then 1:m or m:1	Divided into 1:1, 1:1 etc. then 1:0, m:0, 0:1 or 0:m	Not allowed	All matches all	Divided into 1:1, 1:1 etc. then 1:m or m:1	All matches all	All matches all	All matches all
0:1	Added to output with Right join or Full join	Omitted from output by using In option and If condition	May be omitted from output by using In subcommand and Select If command	Not added to output	Added to output with umatched (master) or umatched (both) option	Omitted from output with the option keep(matched master)	Added to output (right_join and full_join)	Added to output (when argument *all* or *all.y* is TRUE)	Added to output (when argument *how* is right or outer)
0:m	Added to output with Right join or Full join	Omitted from output when using In option and If condition	Omitted from output by using In subcommand and Select If command	Not added to output	Added to output with umatched (master) or umatched (both) option	Omitted from output with the option keep(matched master)	Added to output (right_join and full_join)	Added to output (when argument *all* or *all.y* is TRUE)	Added to output (when argument *how* is right or outer)
1:0	Added to output with Left join or Full join	Omitted from output when using In option and If condition	Omitted from output by using In subcommand and Select If command	Added to output	Added to output with umatched (using) or umatched (both)option	Omitted from output with the option keep(matched using)	Added to output (left_join and full_join)	Added to output (when argument *all* or *all.x* is TRUE)	Added to output (when argument *how* is left or outer)
m:0	Added to output with Left join or Full join	Omitted from output when using In option and If condition	Omitted from output by using In subcommand and Select If command	Added to output	Added to output with umatched (using) or umatched (both) option	Omitted from output with the option keep(matched using)	Added to output (left_join and full_join)	Added to output (when argument *all* or *all.x* is TRUE)	Added to output (when argument *how* is left or outer)

15. Aggregation

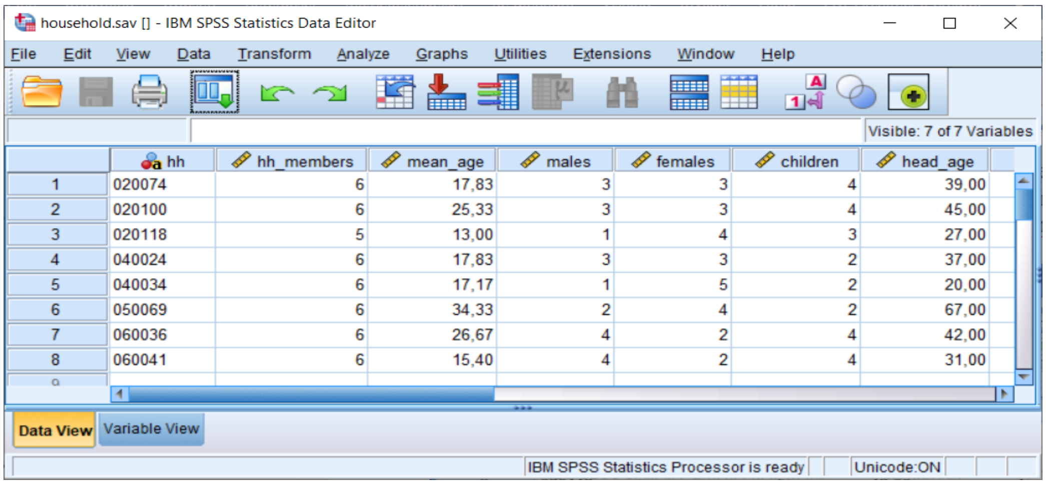
To aggregate a dataset means to group observations with common values for some variables together into one single observation. When we have a dataset with one observation for each member of a household, we can aggregate this dataset to contain one observation for each household instead. All observations with the same household identification will be added together. Here is an example where we want to aggregate to a household dataset and count the number of persons in each household.

Before aggregation (only the first 17 observations are shown):


mdgperson_nodup.sav [] - IBM SPSS Statistics Data Editor
File Edit View Data Transform Analyze Direct Marketing Graphs Utilities Add-ons Window Help
Visible: 8 of 8 Variables
Data View
Variable View
IBM SPSS Statistics Processor is ready

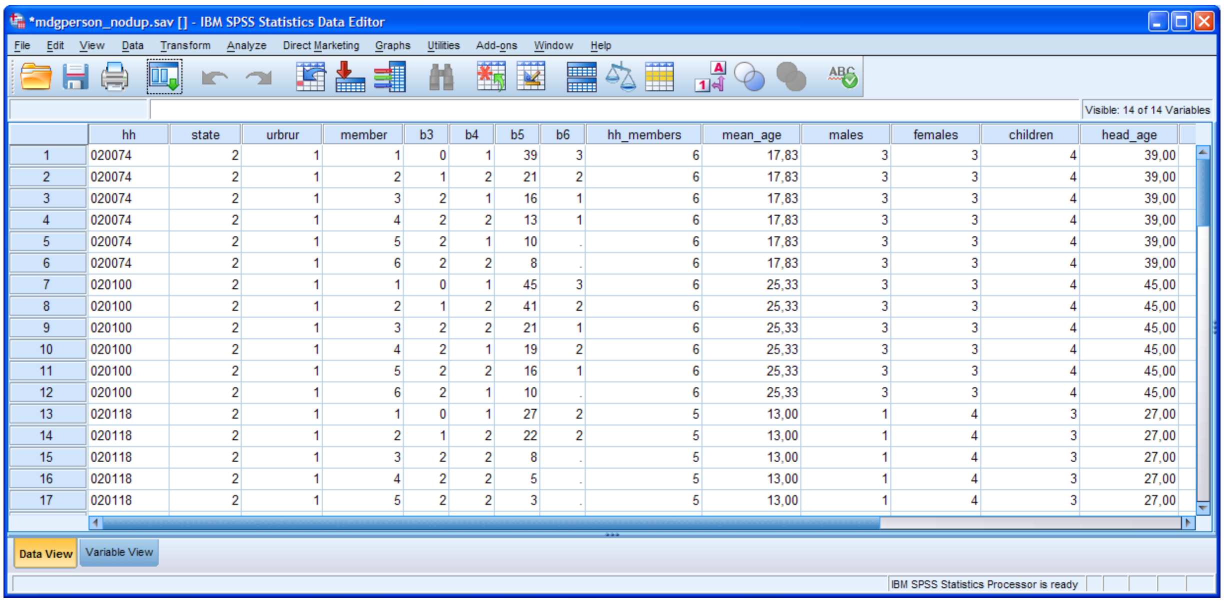
	hh	state	urbrur	member	b3	b4	b5	b6
1	020074	2	1	1	0	1	39	3
2	020074	2	1	2	1	2	21	2
3	020074	2	1	3	2	1	16	1
4	020074	2	1	4	2	2	13	1
5	020074	2	1	5	2	1	10	.
6	020074	2	1	6	2	2	8	.
7	020100	2	1	1	0	1	45	3
8	020100	2	1	2	1	2	41	2
9	020100	2	1	3	2	2	21	1
10	020100	2	1	4	2	1	19	2
11	020100	2	1	5	2	2	16	1
12	020100	2	1	6	2	1	10	.
13	020118	2	1	1	0	1	27	2
14	020118	2	1	2	1	2	22	2
15	020118	2	1	3	2	2	8	.
16	020118	2	1	4	2	2	5	.
17	020118	2	1	5	2	2	3	.

For each household we want to count the number of household members (hh_members), number of males (males), number of females (females), number of children (children), average age of the persons in the household (mean_age) and we want the age of the household head (head_age):


household.sav [] - IBM SPSS Statistics Data Editor
File Edit View Data Transform Analyze Graphs Utilities Extensions Window Help
Visible: 7 of 7 Variables
Data View
Variable View
IBM SPSS Statistics Processor is ready
Unicode:ON

	hh	hh_members	mean_age	males	females	children	head_age
1	020074	6	17,83	3	3	4	39,00
2	020100	6	25,33	3	3	4	45,00
3	020118	5	13,00	1	4	3	27,00
4	040024	6	17,83	3	3	2	37,00
5	040034	6	17,17	1	5	2	20,00
6	050069	6	34,33	2	4	2	67,00
7	060036	6	26,67	4	2	4	42,00
8	060041	6	15,40	4	2	4	31,00

Quite often we want the aggregated variables added to the original data, like this:


*mdgperson_nodup.sav [] - IBM SPSS Statistics Data Editor
File Edit View Data Transform Analyze Direct Marketing Graphs Utilities Add-ons Window Help
Visible: 14 of 14 Variables
Data View
Variable View
IBM SPSS Statistics Processor is ready

	hh	state	urbrur	member	b3	b4	b5	b6	hh_members	mean_age	males	females	children	head_age
1	020074	2	1	1	0	1	39	3	6	17,83	3	3	4	39,00
2	020074	2	1	2	1	2	21	2	6	17,83	3	3	4	39,00
3	020074	2	1	3	2	1	16	1	6	17,83	3	3	4	39,00
4	020074	2	1	4	2	2	13	1	6	17,83	3	3	4	39,00
5	020074	2	1	5	2	1	10	.	6	17,83	3	3	4	39,00
6	020074	2	1	6	2	2	8	.	6	17,83	3	3	4	39,00
7	020100	2	1	1	0	1	45	3	6	25,33	3	3	4	45,00
8	020100	2	1	2	1	2	41	2	6	25,33	3	3	4	45,00
9	020100	2	1	3	2	2	21	1	6	25,33	3	3	4	45,00
10	020100	2	1	4	2	1	19	2	6	25,33	3	3	4	45,00
11	020100	2	1	5	2	2	16	1	6	25,33	3	3	4	45,00
12	020100	2	1	6	2	1	10	.	6	25,33	3	3	4	45,00
13	020118	2	1	1	0	1	27	2	5	13,00	1	4	3	27,00
14	020118	2	1	2	1	2	22	2	5	13,00	1	4	3	27,00
15	020118	2	1	3	2	2	8	.	5	13,00	1	4	3	27,00
16	020118	2	1	4	2	2	5	.	5	13,00	1	4	3	27,00
17	020118	2	1	5	2	2	3	.	5	13,00	1	4	3	27,00

We see that each person in the household has got the aggregated household variables. The values of the household variables are duplicated.

15.1. Sas

We can use *Proc sql* or *Proc means* to make the aggregated dataset. First, we look at aggregation of number of members and average age with *proc sql*:

```
proc sql ;
 create table household as
  select hh, count(hh) as hh_members, mean(b5) as mean_age
  from mdg.mdgperson_nodup
  group by hh
  order by hh;
quit;
```

The same aggregation done with *proc means*:

```
proc means data=mdg.mdgperson_nodup noprint nway missing;
  class hh;
  var b5;
  output out=household (rename=(_freq_=hh_members) drop=_type_)
mean(b5)=mean_age  ;
run;
```

The _freq_ counts the number of observations and is renamed to hh_members. There is also created a variable called _type_. It contains a level indicator for the different combinations of class variables and is not needed in this example because we have excluded all levels except the most detailed with the nway option in the proc means statement.

For the next types of aggregation, *proc sql* is the best one to use. Then we can do some calculations within the aggregation:

```
proc sql ;
 create table household as
  select hh, count(hh) as hh_members, mean(b5) as mean_age,
   N(CASE WHEN b4 = 1 THEN 1 END) as males,
   N(CASE WHEN b4 = 2 THEN 1 END) as females,
   N(CASE WHEN b3 = 2 THEN 1 END) as children,
   Sum(CASE WHEN b3 = 0 THEN b5 END) as head_age
  from mdg.mdgperson_nodup
  group by hh
  order by hh;
quit;
```

To add the household information to each person we change the syntax a little bit. We select all variables from the original dataset with *select ** instead of just selecting the *group by* variable:

```
proc sql ;
 create table person_hh as
  select *, count(hh) as hh_members, mean(b5) as mean_age,
   N(CASE WHEN b4 = 1 THEN 1 END) as males,
   N(CASE WHEN b4 = 2 THEN 1 END) as females,
   N(CASE WHEN b3 = 2 THEN 1 END) as children,
   Sum(CASE WHEN b3 = 0 THEN b5 END) as head_age
  from mdg.mdgperson_nodup
  group by hh
  order by hh;
quit;
```

15.2. Spss

Here we will open the dataset with the *Get* command. Then we do the aggregation with the *Aggregate* command, it is found under the Data > Aggregate menu. Then we open the aggregated dataset with the *Get* command:

```
GET FILE='mdgperson_nodup.sav'.
AGGREGATE
  /OUTFILE='household.sav'
  /BREAK=hh
  /hh_members=N(member)
  /mean_age=MEAN(b5) .
GET FILE='household.sav'.
```

All variables we need must be added before the aggregation. We create new variables with the *If* command. Then we use the *Aggregate* command for the aggregation. Finally, we open the aggregated dataset with the *Get* command:

```
GET  FILE 'mdgperson_nodup.sav' .
IF (b4 = 1) males = 1.
IF (b4 = 2) females = 1.
IF (b3 = 2) children = 1.
IF (b3 = 0) head_age = b5.
EXECUTE.
AGGREGATE
  /OUTFILE='household.sav'
  /BREAK=hh
  /hh_members=N(member)
  /mean_age=MEAN(b5)
  /males=N(males)
  /females=N(females)
  /children=N(children)
  /head_age=MEAN(head_age).
GET FILE='household.sav'.
```

When we want to add the household information to the active dataset, we include the *Mode=addvariables* sub-command to the *Aggregate* command and add a *Save* command (after re-run of the *Get*, *If* and *Execute* commands above):

```
AGGREGATE
  /OUTFILE=* MODE=ADDVARIABLES OVERWRITE=YES
  /BREAK=hh
  /hh_members=N(member)
  /mean_age=MEAN(b5)
  /males=N(males)
  /females=N(females)
  /children=N(children)
  /head_age=MEAN(head_age).
SAVE /OUTFILE='person_hh.sav'.
```

15.3. Stata

We can use the *egen* command with by to add grouped average values to the dataset. We can do it like this:

```
use "mdgperson_nodup", clear
by hh, sort : egen mean_age = mean(b5)
```

However, when we want to add the count, we should switch to the *collapse* command. The *collapse* command is found under the menu Data > Create or change data > Other variable-transformation commands > Make dataset of means, medians, etc., however it is faster to write the syntax:

```
use "mdgperson_nodup", clear
collapse (count) hh_members=member (mean) mean_age=b5 , by(hh)
```

Now we want to add the number of males, females, and the age of head to each observation in the dataset. This is similar to Spss, we create the variables first with the *generate* command and aggregate with the *collapse* command:

```
use "mdgperson_nodup", clear
generate males = 1 if b4 == 1
generate females = 1 if b4 == 2
generate children = 1 if b3 == 2
generate head_age = b5 if b3 == 0
collapse (count) hh_members=member males females children (mean) mean_age=b5
head_age, by(hh)
```

The *collapse* command does not have the possibility to add the aggregated variables to the original dataset. To add these variables, we can use the *merge* command and sort the data again:

```
merge 1:m hh using "mdgperson_nodup.dta"
sort hh member
```

15.4. R

We can use the *dplyr* package to aggregate data frames in R. We group by *hh* and summarise the count and average age.

```
household <- mdgperson_nodup %>%
   group_by(hh) %>%
   summarise(hh_members=n(),mean_age = mean(b5))
```

The result shows that there is a problem with the last household, it has got NA as average age:

```
  hh      hh_members mean_age

  <chr>        <int>    <dbl>

1 020074           6     17.8
2 020100           6     25.3
3 020118           5     13
4 040024           6     17.8
5 040034           6     17.2
6 050069           6     34.3
7 060036           6     26.7
8 060041           6     NA
```

This is because when one value is NA, the result will by default be NA. To avoid this and calculate the average for those who have valid values, we can add the *na.rm=TRUE* argument:

```
household <- mdgperson_nodup %>%
   group_by(hh) %>%
   summarise(hh_members=n(),mean_age = mean(b5, na.rm = TRUE))
```

Now we will have average age for the last household as well:

```
  hh      hh_members mean_age

  <chr>        <int>    <dbl>

1 020074           6     17.8
2 020100           6     25.3
3 020118           5     13
4 040024           6     17.8
5 040034           6     17.2
6 050069           6     34.3
7 060036           6     26.7
8 060041           6     15.4
```

In the next example, we create variables for each category with the *mutate* command and then we do the actual aggregation with *summarise*:

```
household <- mdgperson_nodup %>%
mutate(male=if_else(b4=='Male',1,0),female=if_else(b4=='Female',1,0),child=if
_else(b3=='Daughter/son',1,0),head_age=if_else(b3=='Head',as.numeric(b5),0))
%>%
group_by(hh) %>%
summarise(hh_members=n(),mean_age = mean(b5, na.rm = TRUE),males=sum(male),fe
males=sum(female),children=sum(child),head_age=sum(head_age))
```

Each variable with possible NA's should include the *na.rm = TRUE* option to avoid NA in the aggregated variables:

```
  hh          hh_members mean_age males females children head_age

  <labelled>       <int>    <dbl> <dbl>   <dbl>    <dbl>    <dbl>

1 020074               6     17.8     3       3        4       39
2 020100               6     25.3     3       3        4       45
3 020118               5     13       1       4        3       27
4 040024               6     17.8     3       3        2       37
5 040034               6     17.2     1       5        2       20
6 050069               6     34.3     2       4        2       67
7 060036               6     26.7     4       2        4       42
8 060041               6     15.4     4       2        4       31
```

We can add aggregated data to each row instead of aggregating the data frame. Still, we use *dplyr*. However, we don't use the *summarise* command. Finally, we erase the variables we don't need to store with the *subset* command.

```
person_hh <- mdgperson_nodup %>%
mutate(male=if_else(b4=='Male',1,0),female=if_else(b4=='Female',1,0),child=if
_else(b3=='Daughter/son',1,0),head_a=if_else(b3=='Head',as.numeric(b5),0)) %>
%
group_by(hh) %>%
mutate(hh_members=n(),mean_age = mean(b5, na.rm = TRUE),males=sum(male),femal
es=sum(female),children=sum(child),head_age=sum(head_a)) %>%
subset(select=-c(male,female,child,head_a))
```

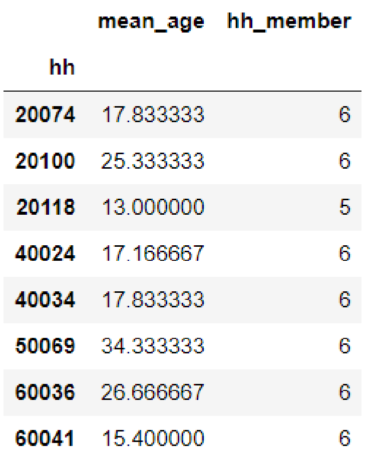
15.5. Python

To aggregate a data frame, we can use the *groupby* function combined with the variables to group by and the variables to aggregate. For the variables we want to aggregate, we also specify what kind of aggregation we want. That can be *count*, *mean*, *sum* and others.

Here we aggregate from persons to households and find the average age of persons and the count of members in the households:

```
mdgperson_nodup.groupby('hh').agg(
    mean_age=('b5', 'mean'),
    hh_member=('member', 'count')
    )
```

The aggregated data:



	mean_age	hh_member
hh		
20074	17.833333	6
20100	25.333333	6
20118	13.000000	5
40024	17.166667	6
40034	17.833333	6
50069	34.333333	6
60036	26.666667	6
60041	15.400000	6

We see above that the column(s) we group by (*hh*) have switched to a row index. Sometimes this is wanted, other times we want to keep the column as it is. To avoid the the column(s) to be a row index, we can add the option as*_index* with the False parameter:

```
mdgperson_nodup.groupby('hh', as_index=False).agg(
    mean_age=('b5', 'mean'),
    hh_member=('member', 'count')
    )
```

Now we see that the group column (*hh*) did not become an index:

	hh	mean_age	hh_member
0	20074	17.833333	6
1	20100	25.333333	6
2	20118	13.000000	5
3	40024	17.166667	6
4	40034	17.833333	6
5	50069	34.333333	6
6	60036	26.666667	6
7	60041	15.400000	6

Sometimes we need to create new columns before we aggregate. Here is an example where we want to count the number of males, females, and children and also the age of the head within each household. The columns for males, females and children are set to 1 when b4 are true and 0 when they are false. The age of the head is set only for persons who are heads, other persons will be given 0 for this variable:

```
mdgperson_nodup['male'] = np.where(mdgperson_nodup['b4'] == "Male", 1, 0)
mdgperson_nodup['female'] = np.where(mdgperson_nodup['b4'] == "Female", 1, 0)
mdgperson_nodup['child'] = np.where(mdgperson_nodup['b3'] == 'Daughter/son',
1, 0)
mdgperson_nodup['head_age'] = np.where(mdgperson_nodup['b3'] == 'Head',
mdgperson_nodup['b5'], 0)
mdgperson_nodup
```

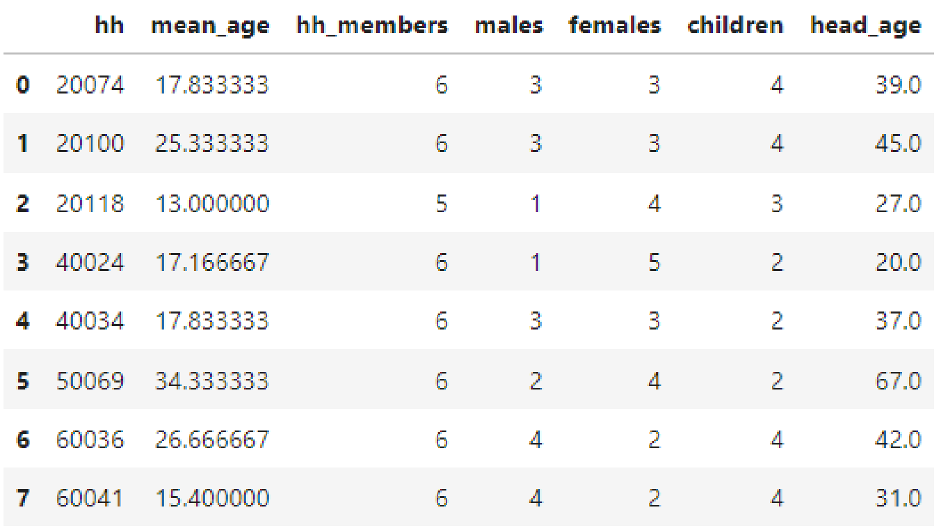
A part of the data frame with the new columns:

	hh	state	urbrur	member	b3	b4	b5	b6	male	female	child	head_age
0	20074	02 Capital	Urban	5	Daughter/son	Male	10.0	missing	1	0	1	0.0
1	20074	02 Capital	Urban	1	Head	Male	39.0	Married - polygamy	1	0	0	39.0
2	60036	06 West	Urban	4	Daughter/son	Male	20.0	Never married	1	0	1	0.0
3	40024	04 East	Urban	1	Head	Female	20.0	Married - polygamy	0	1	0	20.0
4	40024	04 East	Urban	2	Spouse	Female	33.0	Married - monogamy	0	1	0	0.0
5	40034	04 East	Urban	2	Non relative	Female	23.0	Married - polygamy	0	1	0	0.0
6	50069	05 South	Rural	5	Grandchild	Female	16.0	Never married	0	1	0	0.0
7	60036	06 West	Urban	3	Daughter/son	Female	24.0	Never married	0	1	1	0.0
8	20074	02 Capital	Urban	3	Daughter/son	Male	16.0	Never married	1	0	1	0.0
9	50069	05 South	Rural	2	Spouse	Female	60.0	Married - monogamy	0	1	0	0.0

Now we can aggregate to households:

```
mdgperson_nodup.groupby('hh', as_index=False).agg(
    mean_age=('b5', 'mean'),
    hh_members=('member', 'count'),
    males=('male', 'sum'),
    females=('female', 'sum'),
    children=('child', 'sum'),
    head_age=('head_age', 'sum')
    )
```

The aggregated household data frame:

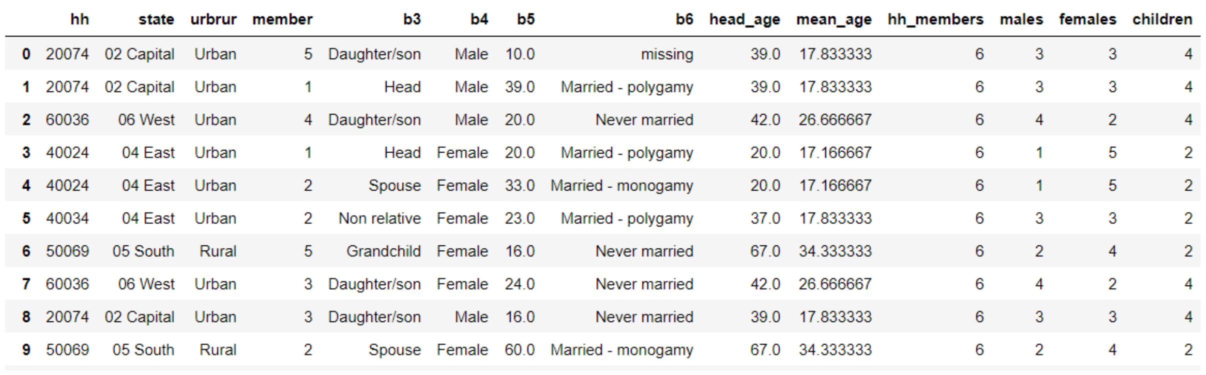


	hh	mean_age	hh_members	males	females	children	head_age
0	20074	17.833333	6	3	3	4	39.0
1	20100	25.333333	6	3	3	4	45.0
2	20118	13.000000	5	1	4	3	27.0
3	40024	17.166667	6	1	5	2	20.0
4	40034	17.833333	6	3	3	2	37.0
5	50069	34.333333	6	2	4	2	67.0
6	60036	26.666667	6	4	2	4	42.0
7	60041	15.400000	6	4	2	4	31.0

When we want to add the aggregated variables to each row in the original data frame, we can add them one at a time. We use the *groupby* function combined with the *transform* function to add aggregated values to each row. Finally, we drop the columns we do not need anymore:

```
mdgperson_nodup['mean_age'] =
mdgperson_nodup.groupby(['hh'])['b5'].transform('mean')
mdgperson_nodup['hh_members'] =
mdgperson_nodup.groupby(['hh'])['member'].transform('count')
mdgperson_nodup['males'] =
mdgperson_nodup.groupby(['hh'])['male'].transform('sum')
mdgperson_nodup['females'] =
mdgperson_nodup.groupby(['hh'])['female'].transform('sum')
mdgperson_nodup['children'] =
mdgperson_nodup.groupby(['hh'])['child'].transform('sum')
mdgperson_nodup['head_age'] =
mdgperson_nodup.groupby(['hh'])['head_age'].transform('sum')
mdgperson_nodup = mdgperson_nodup.drop(columns=['male', 'female', 'child'])
mdgperson_nodup
```

We see that the aggregated variables have the same values within the same *hh* numbers:

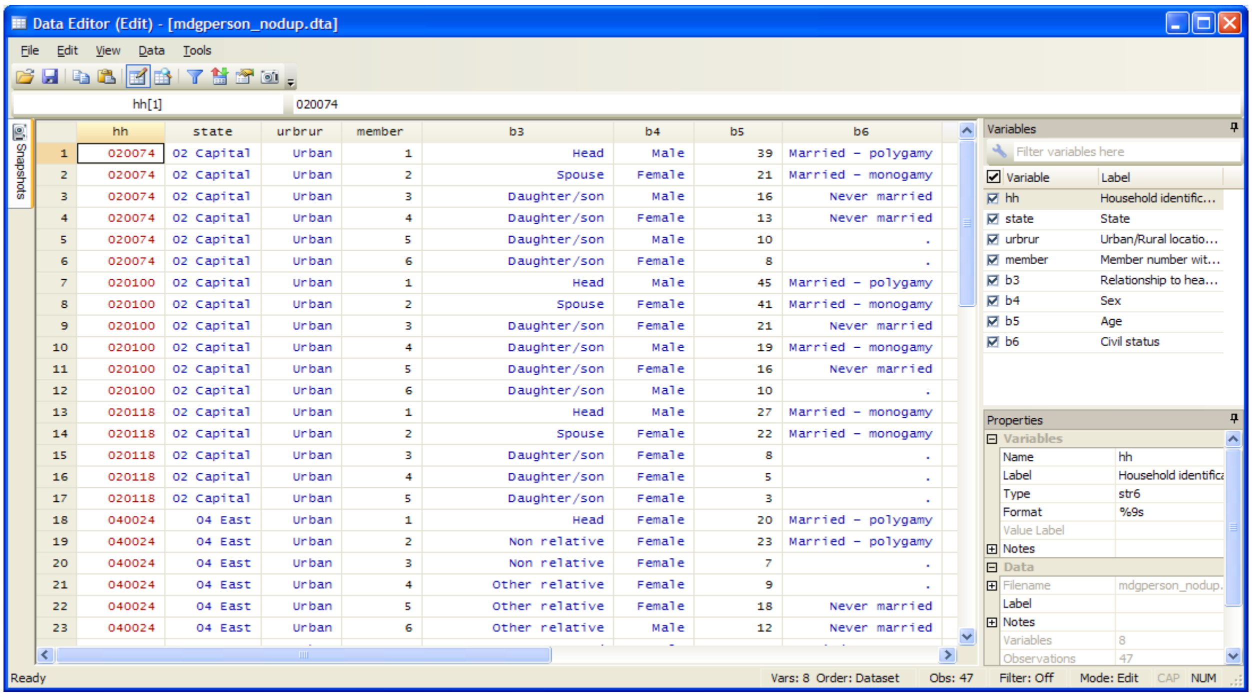


	hh	state	urbrur	member	b3	b4	b5	b6	head_age	mean_age	hh_members	males	females	children
0	20074	02 Capital	Urban	5	Daughter/son	Male	10.0	missing	39.0	17.833333	6	3	3	4
1	20074	02 Capital	Urban	1	Head	Male	39.0	Married - polygamy	39.0	17.833333	6	3	3	4
2	60036	06 West	Urban	4	Daughter/son	Male	20.0	Never married	42.0	26.666667	6	4	2	4
3	40024	04 East	Urban	1	Head	Female	20.0	Married - polygamy	20.0	17.166667	6	1	5	2
4	40024	04 East	Urban	2	Spouse	Female	33.0	Married - monogamy	20.0	17.166667	6	1	5	2
5	40034	04 East	Urban	2	Non relative	Female	23.0	Married - polygamy	37.0	17.833333	6	3	3	2
6	50069	05 South	Rural	5	Grandchild	Female	16.0	Never married	67.0	34.333333	6	2	4	2
7	60036	06 West	Urban	3	Daughter/son	Female	24.0	Never married	42.0	26.666667	6	4	2	4
8	20074	02 Capital	Urban	3	Daughter/son	Male	16.0	Never married	39.0	17.833333	6	3	3	4
9	50069	05 South	Rural	2	Spouse	Female	60.0	Married - monogamy	67.0	34.333333	6	2	4	2

16. Restructuring files

There are many ways to restructure data files. The most common are transposing observations to variables and transposing variables to observations. In our original dataset there is one observation for each person. If we want, we can restructure this dataset to have one set of variables for each person. This will be to transpose from observations to variables. As we see in our data the *b3* variable contains the relation to the head of the household. When we transpose this to one variable for each person it is common to name the new variables b3_1-b3_*n*, where *n* is the maximum number of persons in a household. We want to transpose the variables with person information. The household variables are not to be transposed; they will be kept as they are in the new dataset.

Here is how the dataset looks like before the restructure:



Data Editor (Edit) - [mdgperson_nodup.dta]

hh[1] 020074

	hh	state	urbrur	member	b3	b4	b5	b6
1	020074	02 Capital	Urban	1	Head	Male	39	Married - polygamy
2	020074	02 Capital	Urban	2	Spouse	Female	21	Married - monogamy
3	020074	02 Capital	Urban	3	Daughter/son	Male	16	Never married
4	020074	02 Capital	Urban	4	Daughter/son	Female	13	Never married
5	020074	02 Capital	Urban	5	Daughter/son	Male	10	.
6	020074	02 Capital	Urban	6	Daughter/son	Female	8	.
7	020100	02 Capital	Urban	1	Head	Male	45	Married - polygamy
8	020100	02 Capital	Urban	2	Spouse	Female	41	Married - monogamy
9	020100	02 Capital	Urban	3	Daughter/son	Female	21	Never married
10	020100	02 Capital	Urban	4	Daughter/son	Male	19	Married - monogamy
11	020100	02 Capital	Urban	5	Daughter/son	Female	16	Never married
12	020100	02 Capital	Urban	6	Daughter/son	Male	10	.
13	020118	02 Capital	Urban	1	Head	Male	27	Married - monogamy
14	020118	02 Capital	Urban	2	Spouse	Female	22	Married - monogamy
15	020118	02 Capital	Urban	3	Daughter/son	Female	8	.
16	020118	02 Capital	Urban	4	Daughter/son	Female	5	.
17	020118	02 Capital	Urban	5	Daughter/son	Female	3	.
18	040024	04 East	Urban	1	Head	Female	20	Married - polygamy
19	040024	04 East	Urban	2	Non relative	Female	23	Married - polygamy
20	040024	04 East	Urban	3	Non relative	Female	7	.
21	040024	04 East	Urban	4	Other relative	Female	9	.
22	040024	04 East	Urban	5	Other relative	Female	18	Never married
23	040024	04 East	Urban	6	Other relative	Male	12	Never married

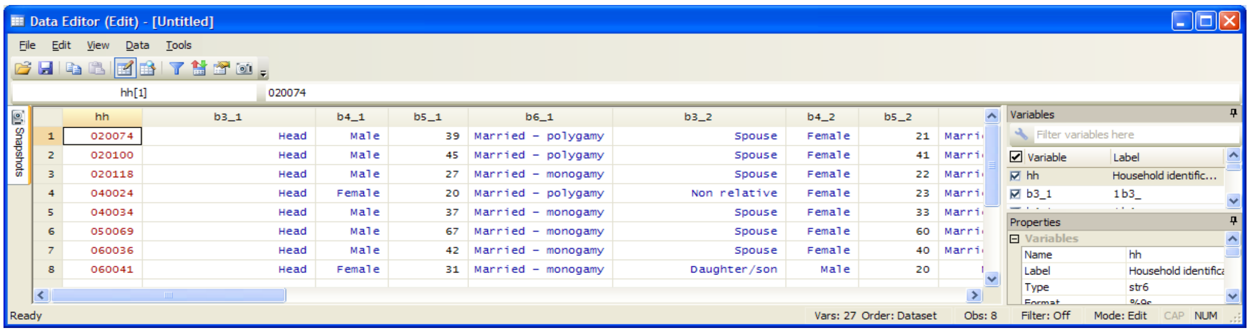
Variable	Label
hh	Household identific...
state	State
urbrur	Urban/Rural locatio...
member	Member number wit...
b3	Relationship to hea...
b4	Sex
b5	Age
b6	Civil status

Properties

Variables	
Name	hh
Label	Household identifica
Type	str6
Format	%9s
Value Label	
Notes	
Data	
Filename	mdgperson_nodup.
Label	
Notes	
Variables	8
Observations	47

Ready Vars: 8 Order: Dataset Obs: 47 Filter: Off Mode: Edit CAP NUM

After the restructure we want it to be like this:



Data Editor (Edit) - [Untitled]

hh[1] 020074

	hh	b3_1	b4_1	b5_1	b6_1	b3_2	b4_2	b5_2	
1	020074	Head	Male	39	Married - polygamy	Spouse	Female	21	Marri
2	020100	Head	Male	45	Married - polygamy	Spouse	Female	41	Marri
3	020118	Head	Male	27	Married - monogamy	Spouse	Female	22	Marri
4	040024	Head	Female	20	Married - polygamy	Non relative	Female	23	Marri
5	040034	Head	Male	37	Married - monogamy	Spouse	Female	33	Marri
6	050069	Head	Male	67	Married - monogamy	Spouse	Female	60	Marri
7	060036	Head	Male	42	Married - monogamy	Spouse	Female	40	Marri
8	060041	Head	Female	31	Married - monogamy	Daughter/son	Male	20	

Variable	Label
hh	Household identific...
b3_1	1 b3_

Properties

Variables	
Name	hh
Label	Household identifica
Type	str6

Ready Vars: 27 Order: Dataset Obs: 8 Filter: Off Mode: Edit CAP NUM

Above, the new variables are shown for the first two persons within each household. Usually this is how our data is structured after data entry. We would like to restructure the data from variables to cases before further data processing.

16.1. Sas

In Sas we have a transpose procedure which can be used. To restructure the data as shown above we restructure each of the b-variables separately and then merge the data together.

```
PROC TRANSPOSE DATA=MDG.MDGPERSON_NODUP OUT=hh_b3 PREFIX=b3_ ;
  BY hh state urbrur;
  ID member;
  VAR b3 ;
RUN;
PROC TRANSPOSE DATA=MDG.MDGPERSON_NODUP OUT=hh_b4 PREFIX=b4_ ;
  BY hh state urbrur;
  ID member;
  VAR b4 ;
RUN;
PROC TRANSPOSE DATA=MDG.MDGPERSON_NODUP OUT=hh_b5 PREFIX=b5_ ;
  BY hh state urbrur;
  ID member;
  VAR b5 ;
RUN;
PROC TRANSPOSE DATA=MDG.MDGPERSON_NODUP OUT=hh_b6 PREFIX=b6_ ;
  BY hh state urbrur;
  ID member;
  VAR b6;
RUN;
data households;
 merge hh_b3 hh_b4 hh_b5 hh_b6;
 by hh state urbrur;
 drop _name_;
run;
```

Each transposed variable is put into a separate dataset together with the identification variables. The prefix option in the *Proc transpose* statement gives the prefix of the new variable names. The *Id* statement is used for the suffixes of the new variable names. The *By* statement defines the identification variables and the *Var* statement tells which variable to transpose. Finally, we merge the datasets together within a Data step. Usually, we should not merge more than two datasets at a time, but as long as we have no duplicates on our person dataset it is safe to merge all five together with one merge.

When we know how to use macros in Sas we can reduce the code to this:

```
%macro ObsToVar(varname);
proc transpose data=mdg.mdgperson_nodup out=hh_&varname. prefix=&varname._ ;
  by hh state urbrur;
  id member;
  var &varname. ;
run;
%mend;
%ObsToVar(b3);
%ObsToVar(b4);
%ObsToVar(b5);
%ObsToVar(b6);
data households;
 merge hh_b3 hh_b4 hh_b5 hh_b6;
 by hh state urbrur;
 drop _name_;
run;
```

The *Proc transpose* procedure is not useful when we change back to the original structure. Instead, we do it with a *Data* step:

```
data persons (keep=hh state urbrur b3 b4 b5 b6 member);
 set households;
 by hh state urbrur ;
 if first.urbrur then
  member = 0;
 array b3a  (*) b3_1-b3_6;
 array b4a  (*) b4_1-b4_6;
 array b5a  (*) b5_1-b5_6;
 array b6a  (*) b6_1-b6_6;
 do i = 1 to dim(b3a);
  member +1;
  b3 = b3a(i);
  b4 = b4a(i);
  b5 = b5a(i);
  b6 = b6a(i);
  if sum(b3,b4,b5,b6) ne . then
   output;
 end;
 label
  b3 = 'Relationship to head of household'
  b4 = 'Sex'
  b5 = 'Age'
  b6 = 'Civil status'
  member = 'Member number within household'
  ;
  format b3 head_rel. b4 sex. b6 civil_status.;
run;
```

In the *Data* statement we name the new dataset and choose which variables to keep. Then we read the dataset households with the *Set* statement and use the *By* statement to define a grouped treatment of the data. We define each group of variables which have the same type of content in separate arrays with *Array* statements. Then we will loop through the arrays. The loop is done within the *Do – End* block. If there is data in any of the b-variables we will output a new observation. Finally, we label the variables and variable values.

16.2. Spss

First, we open the file we will restructure with the *Get* command. Then we use the *Casetovars* command in Spss for our restructure and we can restructure all variables with just one command. The *Casetovars* command is found as a wizard under the Data > Restructure menu and the generated syntax will look like this:

```
GET FILE= 'mdgperson_nodup.sav' .
SORT CASES BY hh state urbrur member.
CASESTOVARS
  /ID=hh state urbrur
  /INDEX=member
  /SEPARATOR='_'
  /GROUPBY=VARIABLE.
```

The *Id* subcommand defines the variables to group by, the *Index* subcommand defines the index where the suffixes for the restructured variable names are found. The rest of the variables will be copied to the new dataset. The subcommand *Groupby* defines how the new variables are to be grouped in the output dataset, either by the original variables (option variable: b3_1 b3_2 .. b3_6, .., b6_1 b6_2 .. b6_6) or by the index variable (option index: b3_1, b4_1, b5_1, b6_1, .. b3_6, b4_6, b5_6, b6_6).

To go back to the original data structure, we use the *Vartocases* command which is also found under the Data > Restructure menu. The generated syntax is this:

```
VARSTOCASES
  /MAKE b3 FROM b3_1 b3_2 b3_3 b3_4 b3_5 b3_6
  /MAKE b4 FROM b4_1 b4_2 b4_3 b4_4 b4_5 b4_6
  /MAKE b5 FROM b5_1 b5_2 b5_3 b5_4 b5_5 b5_6
  /MAKE b6 FROM b6_1 b6_2 b6_3 b6_4 b6_5 b6_6
  /INDEX=member(6)
  /KEEP=hh state urbrur
  /NULL=DROP.
```

If the variables were stored after each other in the dataset, we could use a syntax that choose all the variables from the first one to the last one within the group:

```
b3_1 TO b3_6
```

The syntax above will choose all variables from b3_1 to b3_6 in the order they appear in the dataset. As the variables b3_1 to b3_6 are not stored after each other in our dataset, we must stick to listing all the variables as shown in the *varstocases* example above.

16.3. Stata

The *reshape* command is found under Data > Create or change data > Other variable-transformation commands > Convert data between wide and long. We have a very compact code for doing this in Stata, we just open the dataset to restructure with the *use* command, rename the variables with the *rename* command (to add the underscores) and restructure with the *reshape* command:

```
use "mdgperson_nodup.dta", clear
rename b* b*_
reshape wide b*_, i(hh) j(member)
```

With the *wide* option the structure is changed from observations (long) to variables (wide). The b*_ are the variables to be restructured (= b3_ b4_ b5_ b6_). The *hh* is the by-variable for the restructure; *member* is used to give suffixes to the restructured variable names. The rest of the variables will be copied to the new dataset.

The restructure back to the original form uses the *long* option. After the reshape we rename the b-variables:

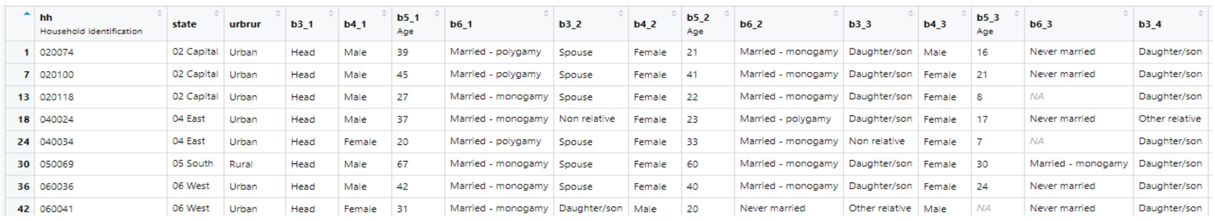
```
reshape long b3_ b4_ b5_ b6_ , i(hh) j(member)
rename b*_ b*
```

16.4. R

We can use the *reshape* command to go from a long to a wide data frame. The *idvar* argument defines the group variables, the *timevar* argument identifies the variable that will be used to differentiate multiple records from the same group, *direction* tells it will be made a wide data frame and *sep* gives the separator between the original variable name and the *timevar* counter.

```
householdsr <- reshape(data.frame(mdgperson_nodup),idvar=c("hh","state","urbr
ur"),timevar="member",direction="wide",sep="_")
```

The new data frame *householdsr* will have one variable for each member in the household for each of the *b3, b4, b5* and *b6* variables where the member number is an index in the variable names. All variables ending with _1 will contain information about member 1, all ending with 2 contain information about member 2 and so on. There is one row for each combination of the variables *hh, state, urbrur* which were the group variables. Here we see some of the variables:

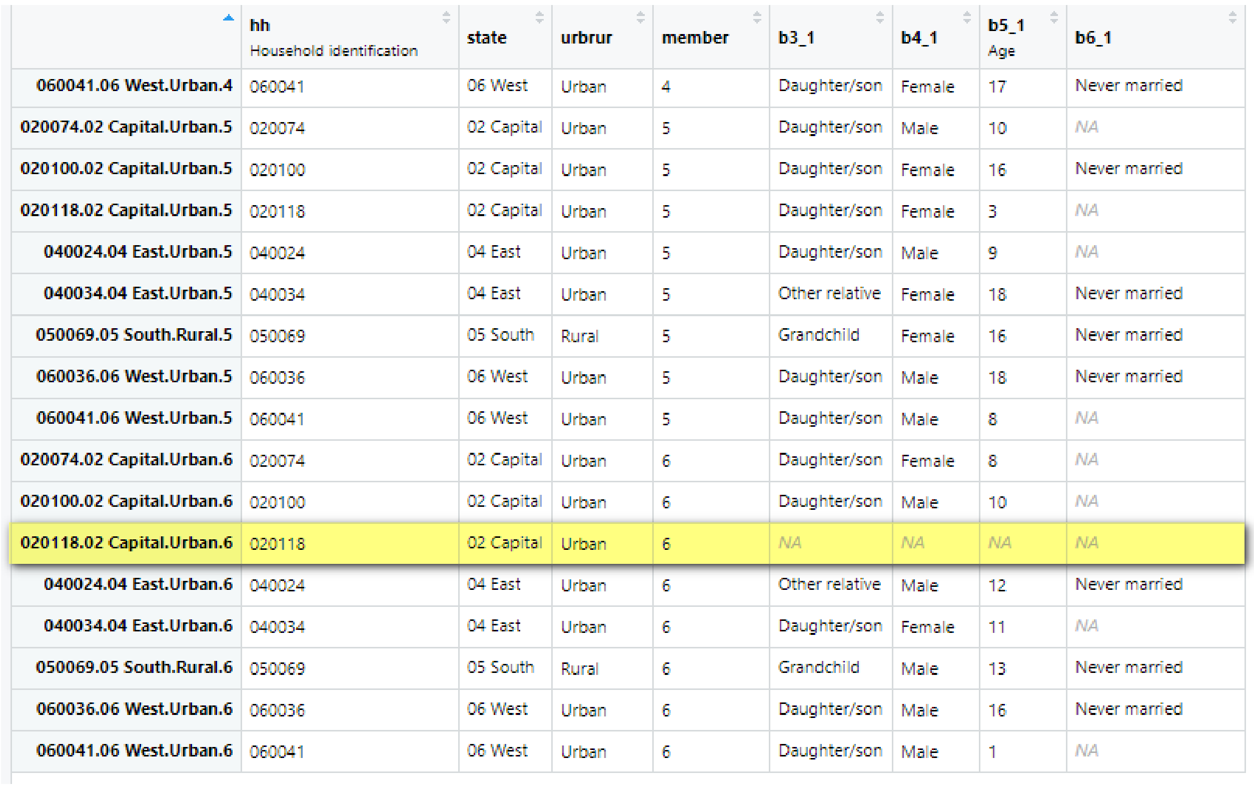


	hh Household identification	state	urbrur	b3_1	b4_1	b5_1 Age	b6_1	b3_2	b4_2	b5_2 Age	b6_2	b3_3	b4_3	b5_3 Age	b6_3	b3_4
1	020074	02 Capital	Urban	Head	Male	39	Married - polygamy	Spouse	Female	21	Married - monogamy	Daughter/son	Male	16	Never married	Daughter/son
7	020100	02 Capital	Urban	Head	Male	45	Married - polygamy	Spouse	Female	41	Married - monogamy	Daughter/son	Female	21	Never married	Daughter/son
13	020118	02 Capital	Urban	Head	Male	27	Married - monogamy	Spouse	Female	22	Married - monogamy	Daughter/son	Female	8	NA	Daughter/son
18	040024	04 East	Urban	Head	Male	37	Married - monogamy	Non relative	Female	23	Married - polygamy	Daughter/son	Female	17	Never married	Other relative
24	040034	04 East	Urban	Head	Female	20	Married - polygamy	Spouse	Female	33	Married - monogamy	Non relative	Female	7	NA	Daughter/son
30	050069	05 South	Rural	Head	Male	67	Married - monogamy	Spouse	Female	60	Married - monogamy	Daughter/son	Female	30	Married - monogamy	Daughter/son
36	060036	06 West	Urban	Head	Male	42	Married - monogamy	Spouse	Female	40	Married - monogamy	Daughter/son	Female	24	Never married	Daughter/son
42	060041	06 West	Urban	Head	Female	31	Married - monogamy	Daughter/son	Male	20	Never married	Other relative	Male	NA	Never married	Daughter/son

When we have our data organised in the wide way, as seen above, we can use reshape to transpose to the long data frame. Then all the variables with the same prefix, like *b3_1-b3_6*, will be transformed to the same variable in separate rows. We use the reshape command again, now with the long *direction*:

```
persons <- reshape(householdsr,idvar=c("hh","state","urbrur"),timevar="member
",direction="long",sep="_")
```

There are some issues with the new, long data frame. The first is that the new variable names end with _1. The second is that there are added some rows where all transposed values are missing (for households which have fewer than the maximum number of household members). The third is that the data frame is not sorted by the *hh* and *member* variables. The fourth is that the row identification is a combination of the group variables and not row numbers:



	hh Household identification	state	urbrur	member	b3_1	b4_1	b5_1 Age	b6_1
060041.06 West.Urban.4	060041	06 West	Urban	4	Daughter/son	Female	17	Never married
020074.02 Capital.Urban.5	020074	02 Capital	Urban	5	Daughter/son	Male	10	NA
020100.02 Capital.Urban.5	020100	02 Capital	Urban	5	Daughter/son	Female	16	Never married
020118.02 Capital.Urban.5	020118	02 Capital	Urban	5	Daughter/son	Female	3	NA
040024.04 East.Urban.5	040024	04 East	Urban	5	Daughter/son	Male	9	NA
040034.04 East.Urban.5	040034	04 East	Urban	5	Other relative	Female	18	Never married
050069.05 South.Rural.5	050069	05 South	Rural	5	Grandchild	Female	16	Never married
060036.06 West.Urban.5	060036	06 West	Urban	5	Daughter/son	Male	18	Never married
060041.06 West.Urban.5	060041	06 West	Urban	5	Daughter/son	Male	8	NA
020074.02 Capital.Urban.6	020074	02 Capital	Urban	6	Daughter/son	Female	8	NA
020100.02 Capital.Urban.6	020100	02 Capital	Urban	6	Daughter/son	Male	10	NA
020118.02 Capital.Urban.6	020118	02 Capital	Urban	6	NA	NA	NA	NA
040024.04 East.Urban.6	040024	04 East	Urban	6	Other relative	Male	12	Never married
040034.04 East.Urban.6	040034	04 East	Urban	6	Daughter/son	Female	11	NA
050069.05 South.Rural.6	050069	05 South	Rural	6	Grandchild	Male	13	Never married
060036.06 West.Urban.6	060036	06 West	Urban	6	Daughter/son	Male	16	Never married
060041.06 West.Urban.6	060041	06 West	Urban	6	Daughter/son	Male	1	NA

We see that the new variables are called *b3_1, b4_1, b5_1* and *b6_1*. We want to delete the suffix _1 in the names. We use the *sub* command with a regular expression and say that the suffix _1 in the column names should be replaced with an empty string:

```
colnames(persons) <- sub("_1$", "", colnames(persons))
```

Next, we will delete the row that has NA's for all the new variables (marked yellow in the list above). We count the NA's for variable 5:8 and keep only those were the count is not 4 (as it is four new variables). The ,5:8 means all rows and columns 5:8. The last comma tell us to use all columns in the data frame in the output.:

```
persons <- persons[rowSums(is.na(persons[,5:8]))!=4,]
```

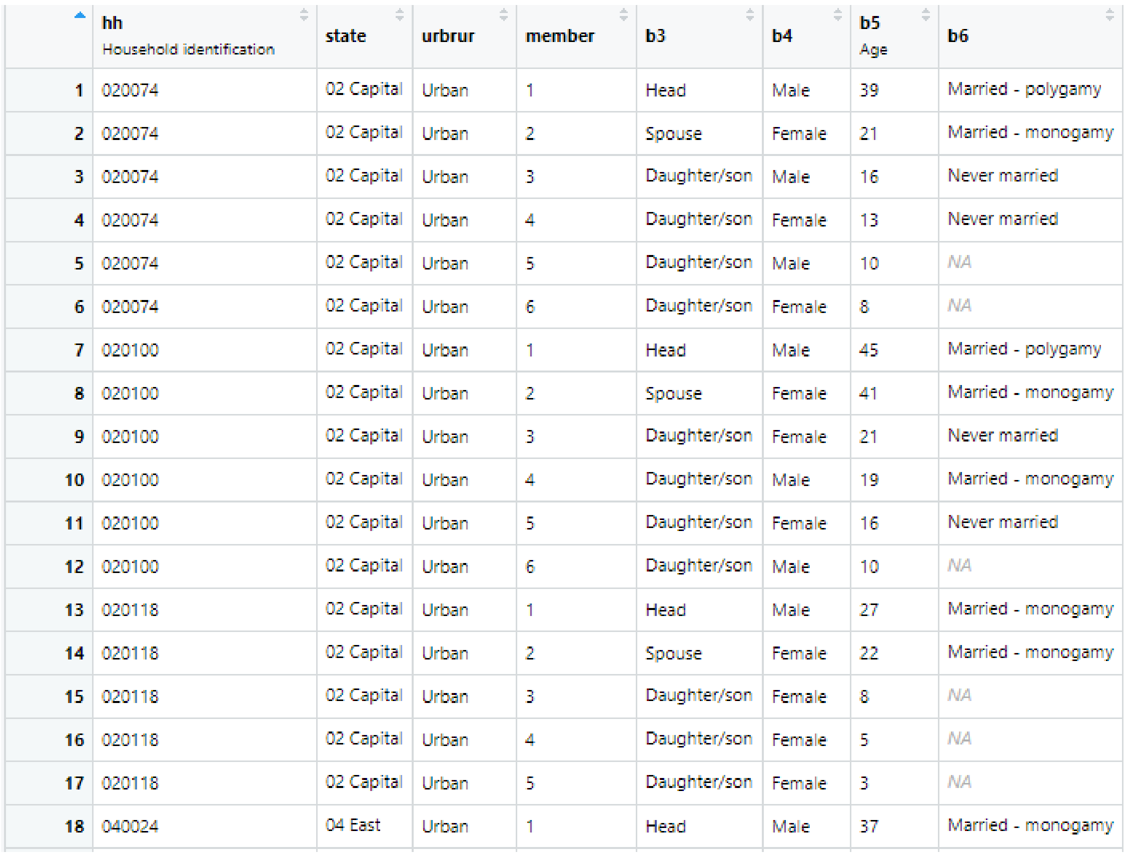
Now we will sort the data frame by the *hh* and *member* variables:

```
persons <- persons %>% arrange(hh,member)
```

Finally, we want to change the row names to row numbers:

```
rownames(persons)<-1:nrow(persons)
```

The data frame is now as we want it to be:



	hh Household identification	state	urbrur	member	b3	b4	b5 Age	b6
1	020074	02 Capital	Urban	1	Head	Male	39	Married - polygamy
2	020074	02 Capital	Urban	2	Spouse	Female	21	Married - monogamy
3	020074	02 Capital	Urban	3	Daughter/son	Male	16	Never married
4	020074	02 Capital	Urban	4	Daughter/son	Female	13	Never married
5	020074	02 Capital	Urban	5	Daughter/son	Male	10	NA
6	020074	02 Capital	Urban	6	Daughter/son	Female	8	NA
7	020100	02 Capital	Urban	1	Head	Male	45	Married - polygamy
8	020100	02 Capital	Urban	2	Spouse	Female	41	Married - monogamy
9	020100	02 Capital	Urban	3	Daughter/son	Female	21	Never married
10	020100	02 Capital	Urban	4	Daughter/son	Male	19	Married - monogamy
11	020100	02 Capital	Urban	5	Daughter/son	Female	16	Never married
12	020100	02 Capital	Urban	6	Daughter/son	Male	10	NA
13	020118	02 Capital	Urban	1	Head	Male	27	Married - monogamy
14	020118	02 Capital	Urban	2	Spouse	Female	22	Married - monogamy
15	020118	02 Capital	Urban	3	Daughter/son	Female	8	NA
16	020118	02 Capital	Urban	4	Daughter/son	Female	5	NA
17	020118	02 Capital	Urban	5	Daughter/son	Female	3	NA
18	040024	04 East	Urban	1	Head	Male	37	Married - monogamy

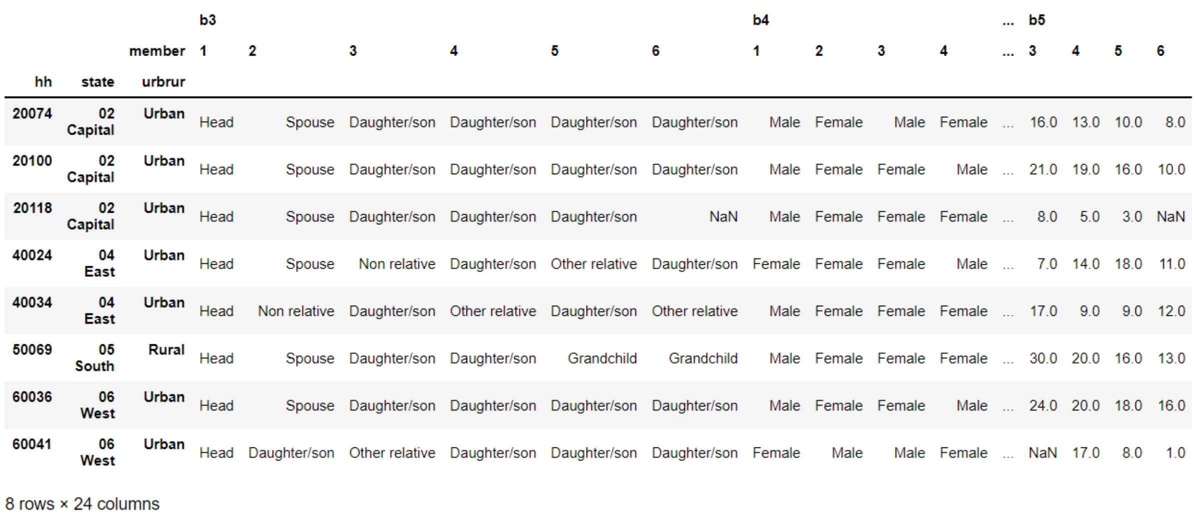
16.5. Python

Our survey, before aggregation, is in the long format. Here that means every person within a household has a separate row in the data frame. If we instead want one row for each household and repeated columns for the persons in the household, we can use the *pivot* function.

We define the variables to keep once for each household in the index parameter. Then we use the columns parameter to specify the person number. Finally, we use the values parameter to specify the columns to pivot:

```
householdsr = mdgperson_nodup.pivot(index=['hh', 'state', 'urbrur'],
columns='member', values= ['b3', 'b4', 'b5', 'b6'])
householdsr
```

The data frame is now in the wide format:



			b3						b4				...	b5			
		member	1	2	3	4	5	6	1	2	3	4	...	3	4	5	6
hh	state	urbrur															
20074	02 Capital	Urban	Head	Spouse	Daughter/son	Daughter/son	Daughter/son	Daughter/son	Male	Female	Male	Female	...	16.0	13.0	10.0	8.0
20100	02 Capital	Urban	Head	Spouse	Daughter/son	Daughter/son	Daughter/son	Daughter/son	Male	Female	Female	Male	...	21.0	19.0	16.0	10.0
20118	02 Capital	Urban	Head	Spouse	Daughter/son	Daughter/son	Daughter/son	NaN	Male	Female	Female	Female	...	8.0	5.0	3.0	NaN
40024	04 East	Urban	Head	Spouse	Non relative	Daughter/son	Other relative	Daughter/son	Female	Female	Female	Male	...	7.0	14.0	18.0	11.0
40034	04 East	Urban	Head	Non relative	Daughter/son	Other relative	Daughter/son	Other relative	Male	Female	Female	Female	...	17.0	9.0	9.0	12.0
50069	05 South	Rural	Head	Spouse	Daughter/son	Daughter/son	Grandchild	Grandchild	Male	Female	Female	Female	...	30.0	20.0	16.0	13.0
60036	06 West	Urban	Head	Spouse	Daughter/son	Daughter/son	Daughter/son	Daughter/son	Male	Female	Female	Male	...	24.0	20.0	18.0	16.0
60041	06 West	Urban	Head	Daughter/son	Other relative	Daughter/son	Daughter/son	Daughter/son	Female	Male	Male	Female	...	NaN	17.0	8.0	1.0

8 rows × 24 columns

Each of the variables *b3, b4, b5* and *b6* has now 6 columns (the maximum number of persons in a household. For households with less than 6 members, the values are set to NaN for numbers higher than the number of persons in the household (see row 3, member 6).

There are some issues with this data frame. First the pivoted columns have multi-index names which should be changed to normal column names. Second, the household variables are defined as an index, they should be ordinary columns. We rename the column names with a for loop through the column names:

```
householdsr.columns = [f'{x}_{y}' for x, y in householdsr.columns]
```

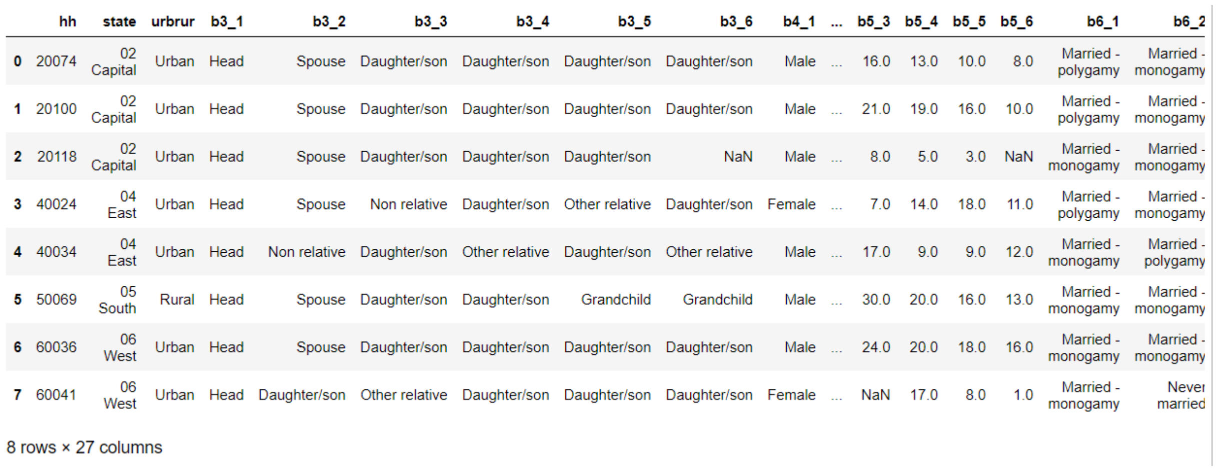
Another way to rename the column names is to use the *get_level_values* function:

```
householdsr.columns = householdsr.columns.get_level_values(0) + '_' +
householdsr.columns.get_level_values(1).astype('str')
```

Then we use the *reset_index* method to change our household variables back to normal columns:

```
householdsr = householdsr.reset_index()
householdsr
```

The result:



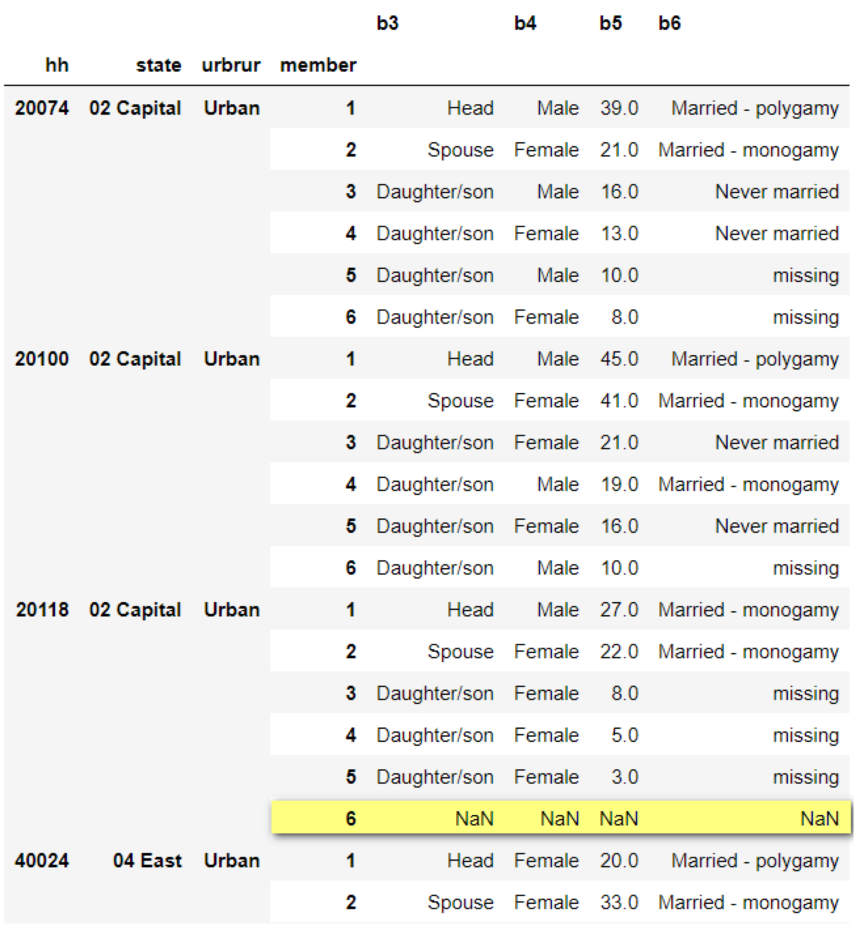
	hh	state	urbrur	b3_1	b3_2	b3_3	b3_4	b3_5	b3_6	b4_1	...	b5_3	b5_4	b5_5	b5_6	b6_1	b6_2
0	20074	02 Capital	Urban	Head	Spouse	Daughter/son	Daughter/son	Daughter/son	Daughter/son	Male	...	16.0	13.0	10.0	8.0	Married - polygamy	Married - monogamy
1	20100	02 Capital	Urban	Head	Spouse	Daughter/son	Daughter/son	Daughter/son	Daughter/son	Male	...	21.0	19.0	16.0	10.0	Married - polygamy	Married - monogamy
2	20118	02 Capital	Urban	Head	Spouse	Daughter/son	Daughter/son	Daughter/son	NaN	Male	...	8.0	5.0	3.0	NaN	Married - monogamy	Married - monogamy
3	40024	04 East	Urban	Head	Spouse	Non relative	Daughter/son	Other relative	Daughter/son	Female	...	7.0	14.0	18.0	11.0	Married - polygamy	Married - monogamy
4	40034	04 East	Urban	Head	Non relative	Daughter/son	Other relative	Daughter/son	Other relative	Male	...	17.0	9.0	9.0	12.0	Married - monogamy	Married - polygamy
5	50069	05 South	Rural	Head	Spouse	Daughter/son	Daughter/son	Grandchild	Grandchild	Male	...	30.0	20.0	16.0	13.0	Married - monogamy	Married - monogamy
6	60036	06 West	Urban	Head	Spouse	Daughter/son	Daughter/son	Daughter/son	Daughter/son	Male	...	24.0	20.0	18.0	16.0	Married - monogamy	Married - monogamy
7	60041	06 West	Urban	Head	Daughter/son	Other relative	Daughter/son	Daughter/son	Daughter/son	Female	...	NaN	17.0	8.0	1.0	Married - monogamy	Never married

8 rows × 27 columns

When we have a data frame in wide format like the one above, we can use the *wide_to_long* function to transpose to the long format. First, we specify the name of the data frame to transpose. Then the variables we want to transpose (*b3, b4, b5, b6*). They are called stub variables. We specify the separator for these stub variables (here: _ because the original variables are named *b3_1, b3_2* and so on). Then we specify the variables to copy with the i parameter. Finally, we number the new rows within each household with the j parameter:

```
persons = pd.wide_to_long(householdsr, ['b3', 'b4', 'b5', 'b6'], sep='_',
i=['hh', 'state', 'urbrur'], j='member')
persons
```

The new, long format data frame:



				b3	b4	b5	b6
hh	**state**	**urbrur**	**member**				
20074	**02 Capital**	**Urban**	**1**	Head	Male	39.0	Married - polygamy
			2	Spouse	Female	21.0	Married - monogamy
			3	Daughter/son	Male	16.0	Never married
			4	Daughter/son	Female	13.0	Never married
			5	Daughter/son	Male	10.0	missing
			6	Daughter/son	Female	8.0	missing
20100	**02 Capital**	**Urban**	**1**	Head	Male	45.0	Married - polygamy
			2	Spouse	Female	41.0	Married - monogamy
			3	Daughter/son	Female	21.0	Never married
			4	Daughter/son	Male	19.0	Married - monogamy
			5	Daughter/son	Female	16.0	Never married
			6	Daughter/son	Male	10.0	missing
20118	**02 Capital**	**Urban**	**1**	Head	Male	27.0	Married - monogamy
			2	Spouse	Female	22.0	Married - monogamy
			3	Daughter/son	Female	8.0	missing
			4	Daughter/son	Female	5.0	missing
			5	Daughter/son	Female	3.0	missing
			6	NaN	NaN	NaN	NaN
40024	**04 East**	**Urban**	**1**	Head	Female	20.0	Married - polygamy
			2	Spouse	Female	33.0	Married - monogamy

We should delete the lines were all the transposed variables are NaN. Furthermore, we can change the household variables from index to normal columns. We use the *dropna* function to drop all rows with NaN for member characteristics. It is important to drop the NaN rows before we reset the index. Otherwise, the household variables will be included in the *dropna* test. They are not Nan, and no rows would then be deleted.

```
persons = persons.dropna(how='all')
persons = persons.reset_index()
persons
```

Here are the first rows our data frame after deletion and index reset:

	hh	state	urbrur	member	b3	b4	b5	b6
0	020074	02 Capital	Urban	1	Head	Male	39.0	Married - polygamy
1	020074	02 Capital	Urban	2	Spouse	Female	21.0	Married - monogamy
2	020074	02 Capital	Urban	3	Daughter/son	Male	16.0	Never married
3	020074	02 Capital	Urban	4	Daughter/son	Female	13.0	Never married
4	020074	02 Capital	Urban	5	Daughter/son	Male	10.0	missing
5	020074	02 Capital	Urban	6	Daughter/son	Female	8.0	missing
6	020100	02 Capital	Urban	1	Head	Male	45.0	Married - polygamy
7	020100	02 Capital	Urban	2	Spouse	Female	41.0	Married - monogamy
8	020100	02 Capital	Urban	3	Daughter/son	Female	21.0	Never married
9	020100	02 Capital	Urban	4	Daughter/son	Male	19.0	Married - monogamy
10	020100	02 Capital	Urban	5	Daughter/son	Female	16.0	Never married
11	020100	02 Capital	Urban	6	Daughter/son	Male	10.0	missing
12	020118	02 Capital	Urban	1	Head	Male	27.0	Married - monogamy
13	020118	02 Capital	Urban	2	Spouse	Female	22.0	Married - monogamy
14	020118	02 Capital	Urban	3	Daughter/son	Female	8.0	missing
15	020118	02 Capital	Urban	4	Daughter/son	Female	5.0	missing
16	020118	02 Capital	Urban	5	Daughter/son	Female	3.0	missing
17	040024	04 East	Urban	1	Head	Female	20.0	Married - polygamy
18	040024	04 East	Urban	2	Spouse	Female	33.0	Married - monogamy

If there are many stub variables the list could be long. Instead of typing them all in, we can put them in a list based on the column names. In our dataset we want to find all variables that ends in an underscore and one or more numbers and extract the name without the underscore and the number. Then we select the rows where the same extracted name appears more than once. Finally, we drop the help variable *idx* and convert to a single list. We can do it like this:

```
stubvars = pd.DataFrame(householdsr.columns)
stubvars = stubvars.replace('_[0-9*]$', '', regex=True)
stubvars['idx'] = stubvars.groupby(0).cumcount()
stubvars = stubvars.loc[stubvars['idx'] == 1]
stubvars = stubvars.drop(columns='idx')
stubvars = stubvars.stack().values.tolist()
stubvars
```

The first line converts the variable names into a data frame called *stubvars*. The second replace the variable names without the extension of underscore and number(s) by using a regular expression that says: find an underscore and one or more numbers from 0-9 at the end of the string and replace it with an empty string. The third line add a column called *idx* which number each row with the same name within the data frame. This means that variables names where the extension is deleted will appear more than once, with different *idx* numbers. The fourth line delete all rows except the once where *idx* equals 1 to get rid of duplicates and names that should not be stub variables. In the fifth line we delete the *idx* column. Finally, in the sixth line we convert the data frame to a single list. To make it single we use the *stack* method. Without using the *stack* method, the list will contain one list for each variable and not fit into the stubnames parameter. The list will look like this and fit:

```
['b3', 'b4', 'b5', 'b6']
```

Now we can use the *stubvars* list in the stubnames parameter:

```
persons = pd.wide_to_long(householdsr, stubnames=stubvars, sep='_', i=['hh',
'state', 'urbrur'], j="member")
persons = persons.dropna(how='all')
persons = persons.reset_index()
persons
```

17. Recoding

The usual recoding is when we want to group values of one variable, for instance create age groups from age. In Spss and Stata we have a separate command for this called *recode*. In Sas we can use the *Select* construction to do the recode if we want to have a new variable on the dataset. In R we can use the *case_when* command in *dplyr*. In Sas, if we just need the recode for tabulation or other listings, we can use a format instead of the recode. When we recode, we must decide what to do with invalid and missing values (more about missing values on page 139).

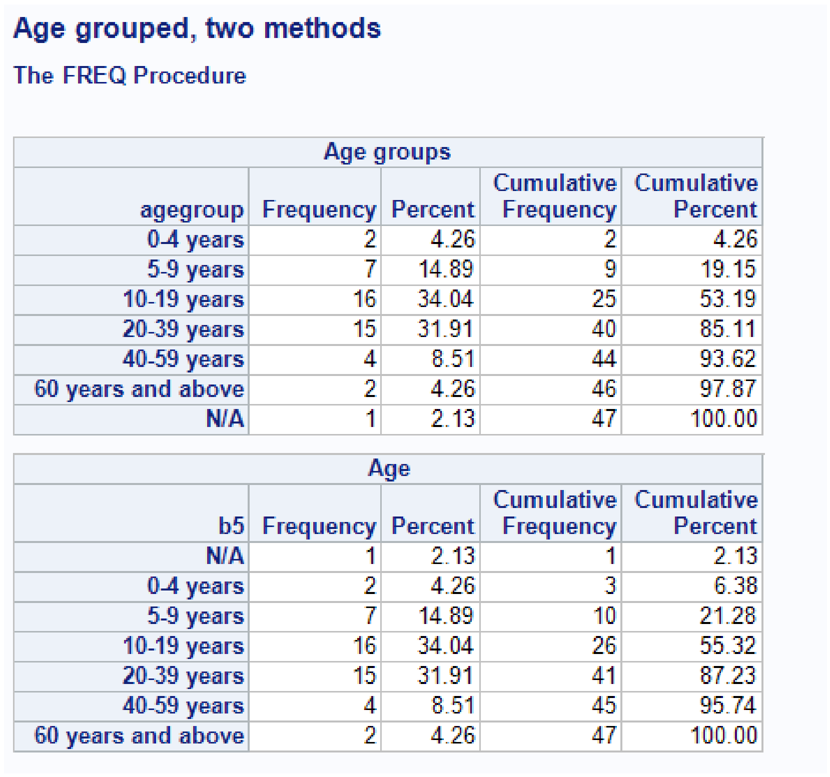
17.1. Sas

Here are two examples on how to recode age into age groups. The first method creates a new variable; the second just uses a format to group values and then uses it in a frequency table. The *Select* construction consists of a number of *When* statements and an *Otherwise* statement. The order of the *When* statements is often important because Sas leaves the *Select* construction after the first true *When* condition is executed.

When we create a new variable, we should also make a format for it. This is done with *Proc format*. We also make the format that group codes together as is shown in the second *Value* statement in *Proc format*. The formats are then loosely connected in the *Freq* procedure:

```
data mdgperson2;
 set mdg.mdgperson_nodup;
 select;
  when ( 0<= b5 <  5) agegroup = 1;
  when ( 5<= b5 < 10) agegroup = 2;
  when (10<= b5 < 20) agegroup = 3;
  when (20<= b5 < 40) agegroup = 4;
  when (40<= b5 < 60) agegroup = 5;
  when (b5 >= 60)     agegroup = 6;
  otherwise           agegroup = 9;
 end;
 label agegroup = 'Age groups';
run;

proc format;
 value agegroup
 1 = '0-4 years'
 2 = '5-9 years'
 3 = '10-19 years'
 4 = '20-39 years'
 5 = '40-59 years'
 6 = '60 years and above'
 9 = 'N/A'
;
 value agegrouped (notsorted)
 0- <  5 = '0-4 years'
 5- < 10 = '5-9 years'
10- < 20 = '10-19 years'
20- < 40 = '20-39 years'
40- < 60 = '40-59 years'
60- high = '60 years and above'
other    = 'N/A'
;
run;

proc freq data=mdgperson2;
 table agegroup b5 /missing;
 format b5 agegrouped. agegroup agegroup.;
 title 'Age grouped, two methods';
run;
```

The result of the two ways of recoding:



Age grouped, two methods

The FREQ Procedure

Age groups				
agegroup	Frequency	Percent	Cumulative Frequency	Cumulative Percent
0-4 years	2	4.26	2	4.26
5-9 years	7	14.89	9	19.15
10-19 years	16	34.04	25	53.19
20-39 years	15	31.91	40	85.11
40-59 years	4	8.51	44	93.62
60 years and above	2	4.26	46	97.87
N/A	1	2.13	47	100.00

Age				
b5	Frequency	Percent	Cumulative Frequency	Cumulative Percent
N/A	1	2.13	1	2.13
0-4 years	2	4.26	3	6.38
5-9 years	7	14.89	10	21.28
10-19 years	16	34.04	26	55.32
20-39 years	15	31.91	41	87.23
40-59 years	4	8.51	45	95.74
60 years and above	2	4.26	47	100.00

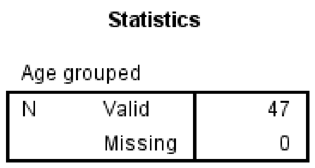
The *notsorted* option in the *Value* statement in *Proc format* is used to tell Sas not to sort the values. The original order is kept. In some of the Sas procedures, like *Proc means* and *Proc tabulate*, we can use this order in the table. *Proc freq* does not have this functionality.

17.2. Spss

In Spss we use the *Recode* command to group the ages, then create the variable and value labels and the frequency table:

```
GET FILE='mdgperson_nodup.sav'.
RECODE b5
 (0 thru 4=1)
 (5 thru 9=2)
 (10 thru 19=3)
 (20 thru 39=4)
 (40 thru 59=5)
 (60 thru Highest=6)
 (SYSMIS=9)
 INTO agegroup.
VARIABLE LABELS agegroup 'Age grouped'.

EXECUTE.
VALUE LABELS agegroup
 1 '0-4 years'
 2 '5-9 years'
 3 '10-19 years'
 4 '20-39 years'
 5 '40-59 years'
 6 '60 years and above'
 9 'N/A'
.
FREQUENCIES agegroup.
```

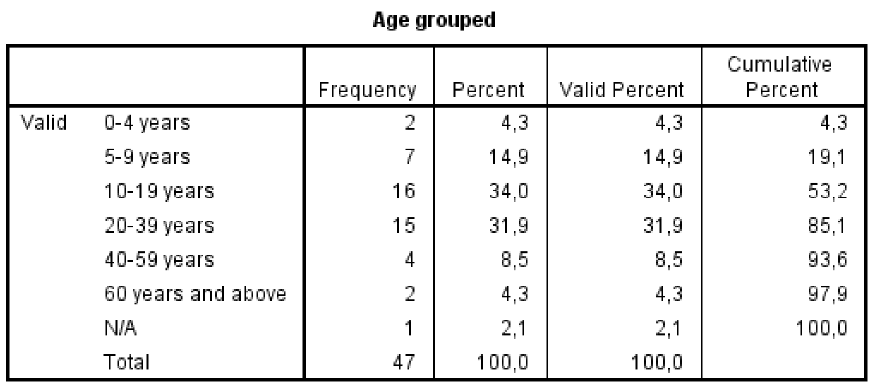
The output gives us the same distribution as in Sas:



Statistics

Age grouped

N	Valid	47
	Missing	0



Age grouped

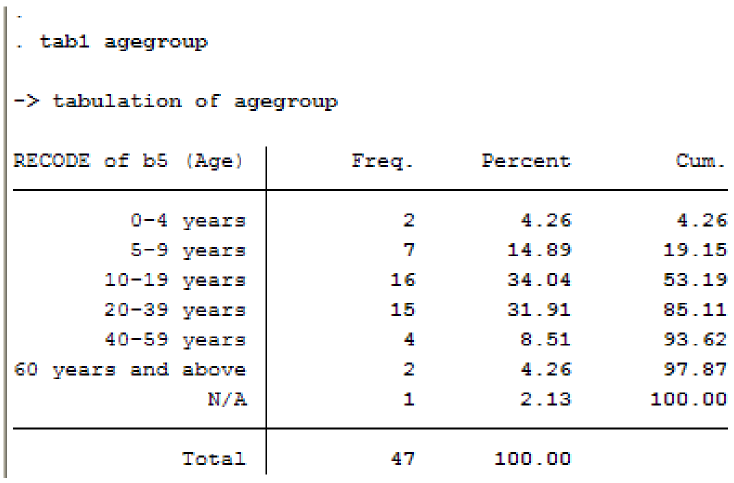
		Frequency	Percent	Valid Percent	Cumulative Percent
Valid	0-4 years	2	4,3	4,3	4,3
	5-9 years	7	14,9	14,9	19,1
	10-19 years	16	34,0	34,0	53,2
	20-39 years	15	31,9	31,9	85,1
	40-59 years	4	8,5	8,5	93,6
	60 years and above	2	4,3	4,3	97,9
	N/A	1	2,1	2,1	100,0
	Total	47	100,0	100,0	

17.3. Stata

The recode in Stata is similar to Spss, except the syntax is even shorter:

```
use "mdgperson_nodup.dta", clear

recode b5 (1/4 = 1) (5/9 = 2) (10/19 = 3) (20/39 = 4) (40/59 = 5) (60/max = 6) ///
 (else = 9), gen(agegroup)

label define agegroup ///
 1 "0-4 years" ///
 2 "5-9 years" ///
 3 "10-19 years" ///
 4 "20-39 years" ///
 5 "40-59 years" ///
 6 "60 years and above" ///
 9 "N/A"

label values agegroup agegroup

tab1 agegroup
```

The table looks like this:



```
.
. tab1 agegroup

-> tabulation of agegroup

RECODE of b5 (Age) |      Freq.     Percent        Cum.
-------------------+-----------------------------------
          0-4 years |          2        4.26        4.26
          5-9 years |          7       14.89       19.15
        10-19 years |         16       34.04       53.19
        20-39 years |         15       31.91       85.11
        40-59 years |          4        8.51       93.62
 60 years and above |          2        4.26       97.87
                N/A |          1        2.13      100.00
-------------------+-----------------------------------
              Total |         47      100.00
```

17.4.R

We can use the *case_when* command in *dplyr* to recode a variable into a new one. First is an example for integer values. We use %in for the intervals for the different age groups. 0:4 is recoded to age group 1 and so on. The last condition, TRUE ~ 7 is to recode values that are not already grouped into the other groups to 7. After the recode, we create levels and labels and add them to the age group variable when we convert it to a factor variable. Finally, we make a frequency table as a data frame. When the table is converted to a data frame the table is transposed as we see in the result. We also name the column in the table command:

```
mdgperson2 <- mdgperson_nodup %>%
   mutate(agegroup=
            case_when(
              b5 %in%  0:4 ~ 1,
              b5 %in%  5:9 ~ 2,
              b5 %in% 10:19 ~ 3,
              b5 %in% 20:39 ~ 4,
              b5 %in% 40:59 ~ 5,
              b5 >=60 ~ 6,
              TRUE ~ 7))
agegroup.codes <- c(1,2,3,4,5,6,7)
agegroup.texts <- c('0-4 years','5-9 years','10-19 years','20-39 years','40-5
9 years','60 years and above','N/A')
mdgperson2$agegroup <- factor(mdgperson2$agegroup,levels = agegroup.codes,lab
els=agegroup.texts)
as.data.frame(addmargins(table(agegroup=mdgperson2$agegroup)))
```

The result:

```
            agegroup Freq
1          0-4 years    2
2          5-9 years    7
3        10-19 years   16
4        20-39 years   15
5        40-59 years    4
6 60 years and above    2
7                N/A    1
8                Sum   47
```

In the previous example we recoded the NA's to agegroup 7. If we want to leave it as NA, we can drop the TRUE ~7 condition. Then we also must remove the NA from the levels and labels when we convert to factor variable. Finally, in the next example the conditions allow decimals:

```
mdgperson2 <- mdgperson_nodup %>%
  mutate(agegroup=
           case_when(
             b5 >= 0 & b5 <5 ~ 1,
             b5 >= 5 & b5 < 10 ~ 2,
             b5 >= 10 & b5 < 20 ~ 3,
             b5 >= 20 & b5 < 40 ~ 4,
             b5 >= 40 & b5 < 60 ~ 5,
             b5 >=60 ~ 6))

agegroup.codes <- c(1,2,3,4,5,6)
agegroup.texts <- c('0-4 years','5-9 years','10-19 years','20-39 years','40-5
9 years','60 years and above')
mdgperson2$agegroup <- factor(mdgperson2$agegroup,levels = agegroup.codes,lab
els=agegroup.texts)
as.data.frame(addmargins(table(agegroup=mdgperson2$agegroup,exclude = NULL)))
```

The first condition, b5 >= 0 & b5 <5, tells us that all values up to 5 (but not 5) should be recoded to 1. The second condition, b5 >= 5 & b5 < 10, include 5. That means there are no values between the end of the interval in the first condition and the start of the interval in the second condition. We can say that the intervals are closed. That is not the case in the first recode example. The first condition, %in% 0-4, ends with the value 4 and the second condition, %in% 5:9, starts with 5. Between 4 and 5 there are room for values like 4.1, 4.5 and 4.9. In the first example these would have been recoded to 7, in the second they will be recoded to 1.

The output table is now like this:

```
            agegroup Freq
1          0-4 years    2
2          5-9 years    7
3        10-19 years   16
4        20-39 years   15
5        40-59 years    4
6 60 years and above    2
7               <NA>    1
8                Sum   47
```

17.5. Python

We can use several ways to recode a column into a new one. One way is to write a *recode* function and then use it with an *apply* method to add the recoded column to the data frame:

```
def age_groups(recodevar):
    if 0 <= recodevar < 5:
        return '1'
    elif 5 <= recodevar < 10:
        return '2'
    elif 10 <= recodevar < 20:
        return '3'
    elif 20 <= recodevar < 40:
        return '4'
    elif 40 <= recodevar < 60:
        return '5'
    elif recodevar >= 60:
        return '6'
    else:
        return '9'
persons['AgeGroup'] = persons['b5'].apply(age_groups)
persons
```

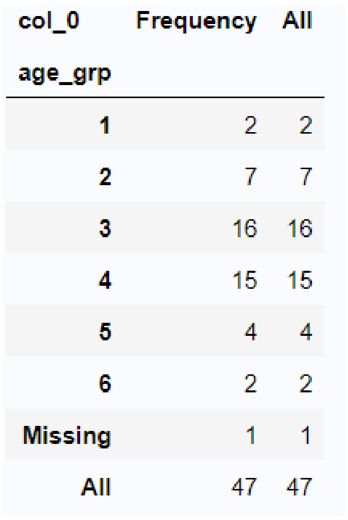
With this recode all rows will get a value for the new *AgeGroup* column, even the NaN values will be recoded as they are included in the else clause.

Another way is to use the *cut* method. However, to be sure the recode is done correctly the column to recode should be a float or integer (integers don't allow NaN's), not object. We define the edge values in one list and the recode values in another list. We can choose to include the rightmost edge values or not in our recode (the *right* parameter). We can also choose whether the first interval should be left-inclusive or not (the *include_lowest* parameter). NaN values will not be recoded with this method.

```
ages = [0, 5, 10, 20, 40, 60, 120]
agegroups=[1, 2, 3, 4, 5, 6]
persons['b5'] = persons['b5'].astype('float64')
persons['age_grp'] = pd.cut(persons['b5'], bins=ages, labels=agegroups,
right=False)
pd.crosstab(persons['age_grp'].astype('object').fillna('Missing'),
columns='Frequency', margins=True)
```

This recode will not include the right edges in the current interval. For instance, the age 20 will be recoded to 4, not 3. The number of agegroups must be one less than the number of age edges. The agegroup 1 will include the age values 0 to 4, the second from 5 to 9 and so on. Values from 120 and up from -1 and down will be Nan. A NaN value for age will result in a NaN value for age_grp.

The crosstab of the recoded age_grp:



col_0	Frequency	All
age_grp		
1	2	2
2	7	7
3	16	16
4	15	15
5	4	4
6	2	2
Missing	1	1
All	47	47

We can fill the missing values with a value after the recode, but then we must change type to object first:

```
persons['age_grp'] = pd.cut(persons['b5'], bins=ages, labels=agegroups,
right=False).astype('object').fillna('9')
pd.crosstab(persons['age_grp'], columns='Frequency', margins=True)
```

Now we have the value 9 for the missing value, just like for the first recode:

col_0	**Frequency**	**All**
age_grp		
1	2	2
2	7	7
3	16	16
4	15	15
5	4	4
6	2	2
9	1	1
All	47	47

18. Functions

Functions help doing our work easier. A function is a set of rules which take one or more arguments and returns an answer. There are lots of functions like arithmetic, character (string), date and time, mathematical, logical, random number, and truncation functions. The functions may differ in how they work between Sas, Spss, Stata, R and Python so we have to be sure how they work before we use them. This is a list of commonly used functions:

Function	**Function type**	**Sas**	**Spss**	**Stata**	**R**	**Python**
Extract a substring	Character	Substr	Substr	substr	substr or substring	str[start:end] (beware that first position is 0 and end is not included)
Concatenate string	Character	Cat, cats, catt or catx	Concat	concat	paste, paste0	+
Replace string values	Character	Translate or tranwrd	Replace	subinstr or subinword	sub (first) or gsub (all)	str.replace
Change string to lowercase letters	Character	Lowcase	Lower	lower	tolower	str.lower
Change string to uppercase letters	Character	Upcase	Upcase	upper	toupper	str.upper
Reverse a string	Character	Reverse	N/A	reverse	stri_reverse (in package stringi)	str[::-1]
Count appearances of a string	Character	Count	N/A	N/A	str_count (in package stringr)	str.count
Position of first occurrence of a string	Character	Find, findc, findw, index, indexc or indexw	Char.index	strpos	str_locate and str_locate_all (in package stringi)	str.find
Position of last occurrence of a string	Character	Findc, indexc or indexw	Char.rindex	strrpos	str_locate and str_locate_all (in package stringi)	str.rfind
Length of a string	Character	Length, lengthc or lengthn	Char.length or length	length	nchar	str.len
Remove leading and trailing blanks	Character	Strip	N/A	N/A	trimws	str.strip
Remove leading characters	Character	Left	Ltrim	Ltrim	trimws (argument which="left")	str.lstrip

Remove trailing blanks	Character	Trim	Rtrim	Rtrim	trimws (argument which="right")	str.rstrip
Fill string with blanks at the left	Character	Right	Char.lpad	N/A	str_pad (in package stringr, argument side="right")	str.pad (with side='left')
Fill string with blanks at the right	Character	Subpad	Char.rpad	N/A	str_pad (in package stringr, argument side="left")	str.pad (with side='right')
Split text into separate words as a list	Character	N/A	N/A	split (into separate variables)	strsplit	str.split
Join text from words in a list	Character	N/A	N/A	N/A	paste (when the list is not in a data frame column)	str.join
Addition	Mathematical	Sum	Sum	rowtotal	rowSums or sum (dplyr)	+
Exponentiation	Mathematical	Exp	Exp	exp	exp	** or np.power
Absolute value	Arithmetic	Abs	Abs	abs	abs	abs
The modulus of a fraction	Arithmetic	Mod	Mod	mod	%%	%
Square root	Arithmetic	Sqrt	Sqrt	sqrt	sqrt	np.sqrt
Check for values	Logical	N/A	Any	strmatch	str_match in stringr package	str.contains
Check for missing value	Logical	Missing	Missing or sysmis	missing	is.na	isna
Coefficient of variation	Statistical	Cv	Cfvar	N/A	cv	N/A (but we can use np.std(x)/np.mean(x))
Maximum value	Statistical	Max	Max	rowmax	max (with rowwise in dplyr package)	np.max
Average value	Statistical	Mean	Mean	rowmean	rowmeans	np.mean
Median value	Statistical	Median	Median	rowmedian	median (in dplyr package)	np.median

Minimum value	Statistical	Min	Min	rowmin	min (with rowwise in dplyr package)	np.min
Number of missing values	Statistical	Nmiss	Nmiss	rowmiss	rowSums(is.na())	isnull in combination with sum
Number with valid, non-missing values	Statistical	N	Nvalid	rownonmiss	rowSums(!is.na( ))	count
Standard deviation	Statistical	Std	Sd	rowsd	sd (in dplyr package)	np.std
Variance	Statistical	Var	Variance	N/A	rowVars	np.var
Round a number	Truncation	Round	Rnd	round	round	round
The integer of a number	Truncation	Int	Trunc	int	integer og as.integer	astype(int)
Random normal distribution	Random number	Normal	Normal	normal	rnorm	np.random.normal
Random uniform distribution	Random number	Uniform	Uniform	runiform	runif	np.random.uniform
Difference between dates	Date and time	Datdif	Datedif	tin or twithin	difftime	N/A (but can use arithmetic difference between 2 date variables)
Convert to date variable	Date and time	Mdy	Date.dmy or Date.mdy	date	as.Date	pd.to_datetime
Convert to time variable	Date and time	Dhms	Time.hms	clock	POSIXct	pd.to_datetime
Return date from a date/time variable	Date and time	Datepart	Xdate.date	dofc or dofC	strftime	dt.date
Return hour from a date/time variable	Date and time	Hour	Xdate.hour	hh or hhC	strftime	dt.hour
Return Julian date from a date/time variable	Date and time	Juldate	Xdate.jday	doy	strftime	dt.strftime('%j')

Return day of month from a date/time variable	Date and time	Day	Xdate.mday	day	strftime	dt.day
Return minute from a date/time variable	Date and time	Minute	Xdate.minute	mm or mmC	strftime	dt.minute
Return month from a date/time variable	Date and time	Month	Xdate.month	month	strftime	dt.month
Return quarter from a date/time variable	Date and time	Qtr	Xdate.quarter	quarter	quarters	dt.quarter
Return second from a date/time variable	Date and time	Second	Xdate.second	ss or ssC	strftime	dt.second
Return time of day from a date/time variable	Date and time	Timepart	Xdate.time	N/A	strftime	dt.time
Return week from a date/time variable	Date and time	Week	Xdate.week	week	strftime	dt.isocalendar().week
Return day of week from a date/time variable	Date and time	Weekday	Xdate.wkday	dow	strftime	dt.weekday
Return year from a date/time variable	Date and time	Year	Xdate.year	year	strftime	dt.year
Fetch a value from a previous observation	Special	Lag	Lag	Use varname[_n - x] instead of a function	lag (in dplyr package)	shift or ffill
Convert string to number	Special	N/A	Number	real	as.numeric	astype(int)
Convert number to string	Special	N/A	String	string	as.character	astype(str)
Convert string/number to factor	N/A	N/A	N/A	N/A	as.factor	astype('category')

Functions are used as part of the programming and may be used in many places.

Below are examples on some of the string functions and how they can be used in each of the softwares. There is also an example of the use of the addition (*Sum in Sas and SPSS, rowtotal in Stata, rowSums in R and + in Python*) function. We also show the difference between adding variables together with sum functions and adding them together the ordinary way (with +).

18.1. Sas

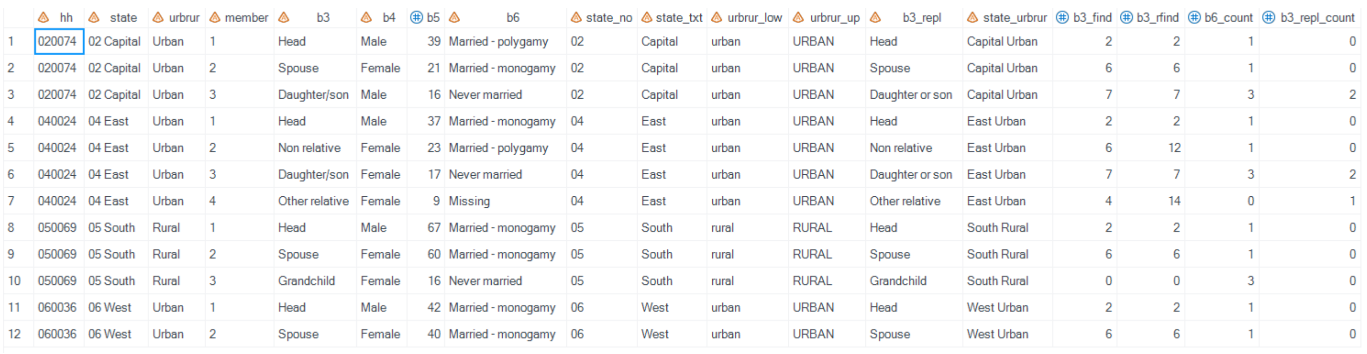
Functions may be used both in data and proc steps. As seen above there are lots of functions to choose from. We will now look at examples of a of these functions. We start with importing an inserted dataset which have full texts in the variables. the texts are not connected as formats because we can't use functions directly on formatted values, only the actual values. Here is a subset of our survey data:

```
data mdgperson_txt;
 infile cards truncover dlm=';';
 input hh : $6.
       state : $10.
       urbrur : $5.
       member : $1.
       b3 : $17.
       b4 : $6.
       b5 : 2.
       b6 : $18.
       ;
cards4;
020074;02 Capital;Urban;1;Head;Male;39;Married - polygamy
020074;02 Capital;Urban;2;Spouse;Female;21;Married - monogamy
020074;02 Capital;Urban;3;Daughter/son;Male;16;Never married
040024;04 East;Urban;1;Head;Male;37;Married - monogamy
040024;04 East;Urban;2;Non relative;Female;23;Married - polygamy
040024;04 East;Urban;3;Daughter/son;Female;17;Never married
040024;04 East;Urban;4;Other relative;Female;9;Missing
050069;05 South;Rural;1;Head;Male;67;Married - monogamy
050069;05 South;Rural;2;Spouse;Female;60;Married - monogamy
050069;05 South;Rural;3;Grandchild;Female;16;Never married
060036;06 West;Urban;1;Head;Male;42;Married - monogamy
060036;06 West;Urban;2;Spouse;Female;40;Married - monogamy
;;;;
run;
```

Now we can use some functions on some of these variables. Note that for the *substr* function we don't need to specify the length of the substring when we want the rest of the string from the start value. We can also nest functions. They are evaluated from inner to outer:

```
data mdgperson_f;
 set mdgperson_txt;
 state_no = substr(state,1,2);
 state_txt = substr(state,4);
 urbrur_low = lowcase(urbrur);
 urbrur_up = upcase(urbrur);
 b3_repl = tranwrd(b3,'/',' or ');
 state_urbrur = catx(' ',substr(state,4),urbrur);
 b3_find = find(b3,'e');
 b3_rfind = findc(b3,'e','b');
 b6_count = count(b6,'e');
 b3_repl_count = count(b3_repl,'r ');
run;
```

The new variables are added as shown here:

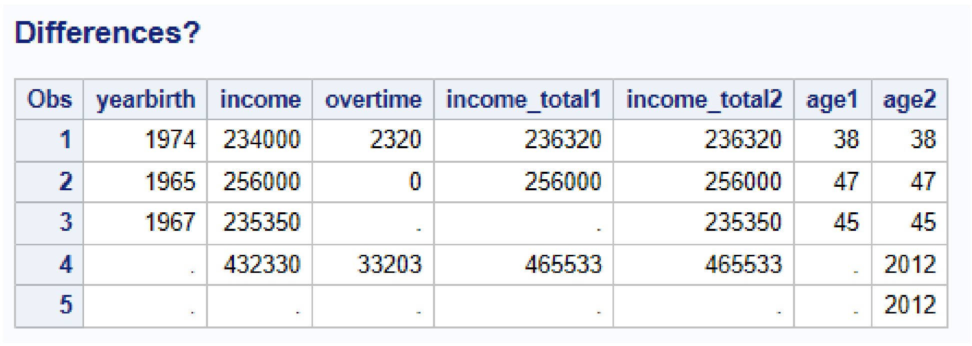


	hh	state	urbrur	member	b3	b4	b5	b6	state_no	state_txt	urbrur_low	urbrur_up	b3_repl	state_urbrur	b3_find	b3_rfind	b6_count	b3_repl_count
1	020074	02 Capital	Urban	1	Head	Male	39	Married - polygamy	02	Capital	urban	URBAN	Head	Capital Urban	2	2	1	0
2	020074	02 Capital	Urban	2	Spouse	Female	21	Married - monogamy	02	Capital	urban	URBAN	Spouse	Capital Urban	6	6	1	0
3	020074	02 Capital	Urban	3	Daughter/son	Male	16	Never married	02	Capital	urban	URBAN	Daughter or son	Capital Urban	7	7	3	2
4	040024	04 East	Urban	1	Head	Male	37	Married - monogamy	04	East	urban	URBAN	Head	East Urban	2	2	1	0
5	040024	04 East	Urban	2	Non relative	Female	23	Married - polygamy	04	East	urban	URBAN	Non relative	East Urban	6	12	1	0
6	040024	04 East	Urban	3	Daughter/son	Female	17	Never married	04	East	urban	URBAN	Daughter or son	East Urban	7	7	3	2
7	040024	04 East	Urban	4	Other relative	Female	9	Missing	04	East	urban	URBAN	Other relative	East Urban	4	14	0	1
8	050069	05 South	Rural	1	Head	Male	67	Married - monogamy	05	South	rural	RURAL	Head	South Rural	2	2	1	0
9	050069	05 South	Rural	2	Spouse	Female	60	Married - monogamy	05	South	rural	RURAL	Spouse	South Rural	6	6	1	0
10	050069	05 South	Rural	3	Grandchild	Female	16	Never married	05	South	rural	RURAL	Grandchild	South Rural	0	0	3	0
11	060036	06 West	Urban	1	Head	Male	42	Married - monogamy	06	West	urban	URBAN	Head	West Urban	2	2	1	0
12	060036	06 West	Urban	2	Spouse	Female	40	Married - monogamy	06	West	urban	URBAN	Spouse	West Urban	6	6	1	0

There is a distinction between the SUM function and addition with + which is important to know. To illustrate this, we first use the *Data* step to create a dataset with some income data. Then, in the same step we calculate total income and age in two different ways. Then we list the dataset with *Proc print*:

```
data test;
  input yearbirth income overtime;
  income_total1 = income+overtime;
  income_total2 = SUM(income,overtime);
  age1   = 2012-yearbirth;
  age2   = SUM(2012,-yearbirth);
cards;
1974 234000 002320
1965 256000      0
1967 235350      .
   . 432330 033203
   .      .      .
;
run;

proc print data=test;
  title 'Differences?' ;
run;
```

The listing of the dataset looks like this:



Differences?

Obs	yearbirth	income	overtime	income_total1	income_total2	age1	age2
1	1974	234000	2320	236320	236320	38	38
2	1965	256000	0	256000	256000	47	47
3	1967	235350	.	.	235350	45	45
4	.	432330	33203	465533	465533	.	2012
5	.	.	.	.	.	.	2012

We see that Sas calculates different results when one or more of the variables we add together are missing. With the *sum* function all non-missing variables are added together. The assumption made in this calculation is that missing values have the value of 0 (except if all variables have missing values, see observation 5). If that is not what we want, we can use ordinary addition instead. Then Sas will not calculate a sum unless all added variables have valid values. The assumption now is that because we don't have enough information, we can't calculate a sum. This last assumption is correct when it comes to calculating the age. However, when we calculate the total income, which method to use is not given.

18.2. Spss

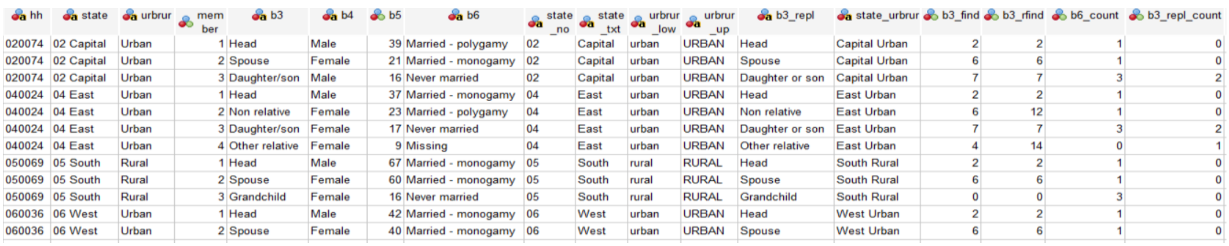
The functions in Spss work similar to Sas. We use them in the *Compute* command. The functions work on the values, not the value labels. Here is a program that import variables where the text is the values so that we can use functions on them:

```
DATA LIST LIST (";")/
       hh (a6)
       state (a10)
       urbrur (a5)
       member (f1)
       b3 (a17)
       b4 (a6)
       b5 (f2)
       b6 (a18)
       .
BEGIN DATA
020074;02 Capital;Urban;1;Head;Male;39;Married - polygamy
020074;02 Capital;Urban;2;Spouse;Female;21;Married - monogamy
020074;02 Capital;Urban;3;Daughter/son;Male;16;Never married
040024;04 East;Urban;1;Head;Male;37;Married - monogamy
040024;04 East;Urban;2;Non relative;Female;23;Married - polygamy
040024;04 East;Urban;3;Daughter/son;Female;17;Never married
040024;04 East;Urban;4;Other relative;Female;9;Missing
050069;05 South;Rural;1;Head;Male;67;Married - monogamy
050069;05 South;Rural;2;Spouse;Female;60;Married - monogamy
050069;05 South;Rural;3;Grandchild;Female;16;Never married
060036;06 West;Urban;1;Head;Male;42;Married - monogamy
060036;06 West;Urban;2;Spouse;Female;40;Married - monogamy
END DATA.
```

Now we use the same functions as in the Sas example above. Beware that there is no function in Spss the counts the number of occurrences of a string in a text. Instead, we can use the length and replace functions to count. It is done by first finding the total length of the variable. Then we subtract the length of the variable where all occurrences of our string are replaced with nothing. This will work when we count only one character. If we count more than one character, we must divide the result by the number of characters we search for, see example below:

```
STRING state_no (a2) state_txt (a7) urbrur_low (a5) urbrur_up (a5) b3_repl
(a17) state_urbrur (a14).
COMPUTE state_no = substr(state,1,2).
COMPUTE state_txt = substr(state,4).
COMPUTE urbrur_low = lower(urbrur).
COMPUTE urbrur_up = upcase(urbrur).
COMPUTE state_urbrur = concat(substr(state,4),' ',urbrur).
COMPUTE b3_find = char.index(b3,'e').
COMPUTE b3_rfind = char.rindex(b3,'e').
COMPUTE b3_repl = replace(b3,'/',' or ').
COMPUTE b6_count = length(b6)-length(replace(b6,'e','')).
COMPUTE b3_repl_count = (length(b3_repl)-length(replace(b3_repl,'r ','')))/2.
EXECUTE.
```

The new variables as they are added to the dataset:



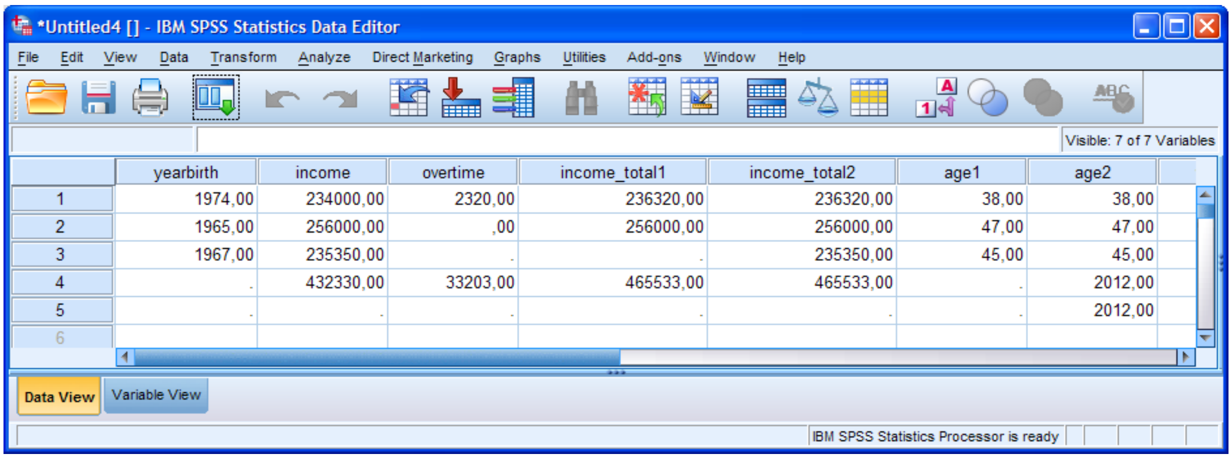
hh	state	urbrur	member	b3	b4	b5	b6	state_no	state_txt	urbrur_low	urbrur_up	b3_repl	state_urbrur	b3_find	b3_rfind	b6_count	b3_repl_count
020074	02 Capital	Urban	1	Head	Male	39	Married - polygamy	02	Capital	urban	URBAN	Head	Capital Urban	2	2	1	0
020074	02 Capital	Urban	2	Spouse	Female	21	Married - monogamy	02	Capital	urban	URBAN	Spouse	Capital Urban	6	6	1	0
020074	02 Capital	Urban	3	Daughter/son	Male	16	Never married	02	Capital	urban	URBAN	Daughter or son	Capital Urban	7	7	3	2
040024	04 East	Urban	1	Head	Male	37	Married - monogamy	04	East	urban	URBAN	Head	East Urban	2	2	1	0
040024	04 East	Urban	2	Non relative	Female	23	Married - polygamy	04	East	urban	URBAN	Non relative	East Urban	6	12	1	0
040024	04 East	Urban	3	Daughter/son	Female	17	Never married	04	East	urban	URBAN	Daughter or son	East Urban	7	7	3	2
040024	04 East	Urban	4	Other relative	Female	9	Missing	04	East	urban	URBAN	Other relative	East Urban	4	14	0	1
050069	05 South	Rural	1	Head	Male	67	Married - monogamy	05	South	rural	RURAL	Head	South Rural	2	2	1	0
050069	05 South	Rural	2	Spouse	Female	60	Married - monogamy	05	South	rural	RURAL	Spouse	South Rural	6	6	1	0
050069	05 South	Rural	3	Grandchild	Female	16	Never married	05	South	rural	RURAL	Grandchild	South Rural	0	0	3	0
060036	06 West	Urban	1	Head	Male	42	Married - monogamy	06	West	urban	URBAN	Head	West Urban	2	2	1	0
060036	06 West	Urban	2	Spouse	Female	40	Married - monogamy	06	West	urban	URBAN	Spouse	West Urban	6	6	1	0

The sum function and addition in Spss work similar to Sas. Here is the syntax in Spss to illustrate the differences:

```
DATA LIST FREE/
 yearbirth income overtime .
BEGIN DATA
1974 234000 002320
1965 256000      0
1967 235350      .
   . 432330 033203
   .      .      .
END DATA.
COMPUTE income_total1 = income+overtime.
COMPUTE income_total2 = SUM(income,overtime).
COMPUTE age1  = 2012-yearbirth.
COMPUTE age2  = SUM(2012,-yearbirth).
EXECUTE.
```

We create the Spss dataset with the *Data list* command, do the calculations with *Compute* commands and finally execute them with the *Execute* command.

Spss also have the same distinction between the normal addition and using the *sum* function, as this output shows:


*Untitled4 [] - IBM SPSS Statistics Data Editor
File Edit View Data Transform Analyze Direct Marketing Graphs Utilities Add-ons Window Help
Visible: 7 of 7 Variables
Data View
Variable View
IBM SPSS Statistics Processor is ready

	yearbirth	income	overtime	income_total1	income_total2	age1	age2
1	1974,00	234000,00	2320,00	236320,00	236320,00	38,00	38,00
2	1965,00	256000,00	,00	256000,00	256000,00	47,00	47,00
3	1967,00	235350,00	.	.	235350,00	45,00	45,00
4	.	432330,00	33203,00	465533,00	465533,00	.	2012,00
5	.	.	.	.	.	.	2012,00
6							

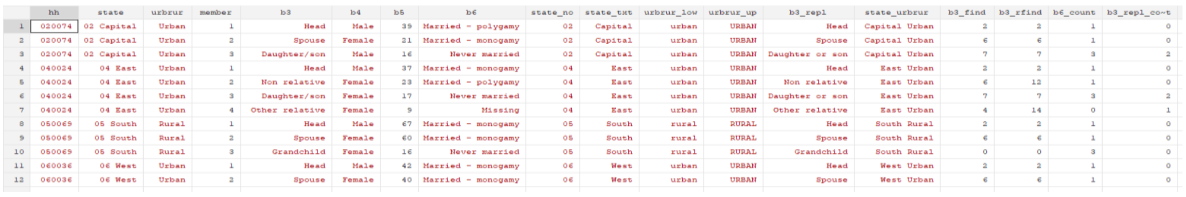
18.3. Stata

The different functions in Stata work either horizontally or vertically. Some functions are used with the *generate* (*gen*) command and some work with the *egen* command. There is no function to count the number of occurrences of a string in a text, but we can do it by combining the *length* and *subinstr* functions similar to Spss. As in Sas and Spss, Stata operates on the actual values for the variables, not the value labels. Hence, in our example we import a test dataset with the value labels as values and then calculate the new variables by using different functions:

```
clear
input str6 hh str10 state str5 urbrur member str17 b3 str6 b4 b5 str18 b6
```

```
020074 "02 Capital" Urban 1 "Head" Male 39 "Married - polygamy"
020074 "02 Capital" Urban 2 "Spouse" Female 21 "Married - monogamy"
020074 "02 Capital" Urban 3 "Daughter/son" Male 16 "Never married"
040024 "04 East" Urban 1 "Head" Male   37 "Married - monogamy"
040024 "04 East" Urban 2 "Non relative" Female 23 "Married - polygamy"
040024 "04 East" Urban 3 "Daughter/son" Female 17 "Never married"
040024 "04 East" Urban 4 "Other relative" Female 9 "Missing"
050069 "05 South" Rural 1 "Head" Male   67 "Married - monogamy"
050069 "05 South" Rural 2 "Spouse" Female 60 "Married - monogamy"
050069 "05 South" Rural 3 "Grandchild" Female 16 "Never married"
060036 "06 West" Urban 1 "Head" Male 42 "Married - monogamy"
060036 "06 West" Urban 2 "Spouse" Female 40 "Married - monogamy"
end

gen state_no = substr(state,1,2)
gen state_txt = substr(state,4,.)
gen urbrur_low = lower(urbrur)
gen urbrur_up = upper(urbrur)
gen b3_repl = subinstr(b3,"/"," or ",.)
egen state_urbrur = concat(state_txt urbrur), punct(" ")
gen b3_find = strpos(b3,"e")
gen b3_rfind = strrpos(b3,"e")
gen b6_count = length(b6) - length(subinstr(b6, "e", "", .))
gen b3_repl_count = (length(b3_repl) - length(subinstr(b3_repl, "r ", "", .)))/2
```

The new variables are added to our dataset:

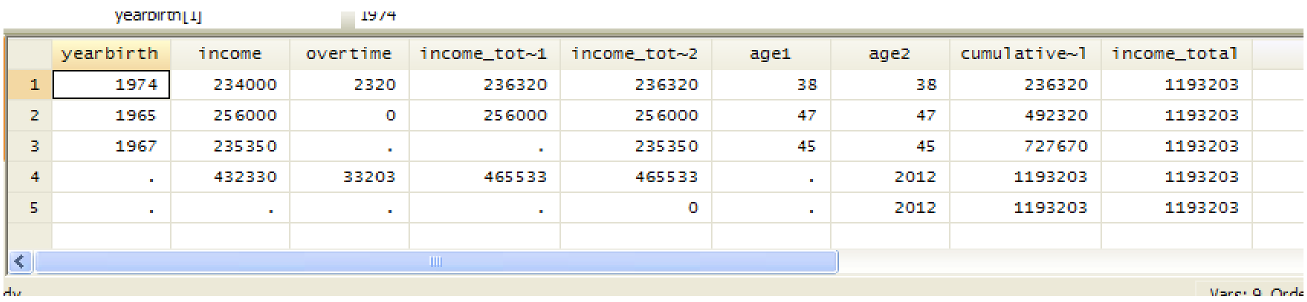


	hh	state	urbrur	member	b3	b4	b5	b6	state_no	state_txt	urbrur_low	urbrur_up	b3_repl	state_urbrur	b3_find	b3_rfind	b6_count	b3_repl_co~t
1	020074	02 Capital	Urban	1	Head	Male	39	Married - polygamy	02	Capital	urban	URBAN	Head	Capital Urban	2	2	1	0
2	020074	02 Capital	Urban	2	Spouse	Female	21	Married - monogamy	02	Capital	urban	URBAN	Spouse	Capital Urban	6	6	1	0
3	020074	02 Capital	Urban	3	Daughter/son	Male	16	Never married	02	Capital	urban	URBAN	Daughter or son	Capital Urban	7	7	3	2
4	040024	04 East	Urban	1	Head	Male	37	Married - monogamy	04	East	urban	URBAN	Head	East Urban	2	2	1	0
5	040024	04 East	Urban	2	Non relative	Female	23	Married - polygamy	04	East	urban	URBAN	Non relative	East Urban	6	12	1	0
6	040024	04 East	Urban	3	Daughter/son	Female	17	Never married	04	East	urban	URBAN	Daughter or son	East Urban	7	7	3	2
7	040024	04 East	Urban	4	Other relative	Female	9	Missing	04	East	urban	URBAN	Other relative	East Urban	4	14	0	1
8	050069	05 South	Rural	1	Head	Male	67	Married - monogamy	05	South	rural	RURAL	Head	South Rural	2	2	1	0
9	050069	05 South	Rural	2	Spouse	Female	60	Married - monogamy	05	South	rural	RURAL	Spouse	South Rural	6	6	1	0
10	050069	05 South	Rural	3	Grandchild	Female	16	Never married	05	South	rural	RURAL	Grandchild	South Rural	0	0	3	0
11	060036	06 West	Urban	1	Head	Male	42	Married - monogamy	06	West	urban	URBAN	Head	West Urban	2	2	1	0
12	060036	06 West	Urban	2	Spouse	Female	40	Married - monogamy	06	West	urban	URBAN	Spouse	West Urban	6	6	1	0

There are differences between using functions and normal addition in Stata as well. We start with clearing the data editor with the *clear* command and creating the dataset with the *input* command. Then we generate the new variables with *generate, egen* and *replace* commands. Here is a syntax that creates the variables as above and some vertically calculated variables too:

```
clear
input yearbirth income overtime
1974 234000 002320
1965 256000      0
1967 235350      .
   . 432330 033203
   .      .      .
end
generate income_total1 = income+overtime
egen income_total2 = rowtotal(income overtime)
generate age1  = 2012-yearbirth
generate year = -2012
egen age2  = rowtotal(year yearbirth)
replace age2 = abs(age2)
drop year
generate cumulative_income_total = sum(income_total2)
egen income_total = total(income_total2)
```

The output looks like this:



yearbirth[1] 1974

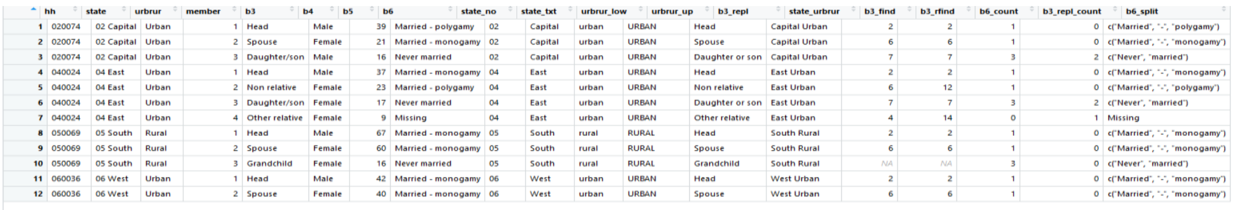
	yearbirth	income	overtime	income_tot~1	income_tot~2	age1	age2	cumulative~l	income_total
1	1974	234000	2320	236320	236320	38	38	236320	1193203
2	1965	256000	0	256000	256000	47	47	492320	1193203
3	1967	235350	.	.	235350	45	45	727670	1193203
4	.	432330	33203	465533	465533	.	2012	1193203	1193203
5	.	.	.	.	0	.	2012	1193203	1193203

18.4. R

As in Stata, many functions may work both horizontal and vertical. The way they work depend on the arguments to the functions. There are two libraries with useful string functions we can use, *string* and *stringi*. We first activate them with the *library* command. Then we import our file into an R data frame as an inserted file. Finally, we use the different string functions to calculate new variables:

```
library(stringr)
library(stringi)
mdgperson_txt <- read.csv(sep=";",header=FALSE,
                 col.names=c("hh","state","urbrur","member","b3","b4","b5","b
6"),
colClasses=c("character","character","character","numeric","character","chara
cter","numeric","character"),
                 text="
020074;02 Capital;Urban;1;Head;Male;39;Married - polygamy
020074;02 Capital;Urban;2;Spouse;Female;21;Married - monogamy
020074;02 Capital;Urban;3;Daughter/son;Male;16;Never married
040024;04 East;Urban;1;Head;Male;37;Married - monogamy
040024;04 East;Urban;2;Non relative;Female;23;Married - polygamy
040024;04 East;Urban;3;Daughter/son;Female;17;Never married
040024;04 East;Urban;4;Other relative;Female;9;Missing
050069;05 South;Rural;1;Head;Male;67;Married - monogamy
050069;05 South;Rural;2;Spouse;Female;60;Married - monogamy
050069;05 South;Rural;3;Grandchild;Female;16;Never married
060036;06 West;Urban;1;Head;Male;42;Married - monogamy
060036;06 West;Urban;2;Spouse;Female;40;Married - monogamy")

mdgperson_txt$state_no = substr(mdgperson_txt$state,1,2)
mdgperson_txt$state_txt = substring(mdgperson_txt$state,4)
mdgperson_txt$urbrur_low = tolower(mdgperson_txt$urbrur)
mdgperson_txt$urbrur_up = toupper(mdgperson_txt$urbrur)
mdgperson_txt$b3_repl = gsub("/"," or ",mdgperson_txt$b3)
mdgperson_txt$state_urbrur = paste(mdgperson_txt$state_txt,mdgperson_txt$urbr
ur)
mdgperson_txt$b3_find = str_locate(mdgperson_txt$b3,"e")
mdgperson_txt$b3_rfind = stri_locate_last(mdgperson_txt$b3,fixed="e")
mdgperson_txt$b6_count = str_count(mdgperson_txt$b6,"e")
mdgperson_txt$b3_repl_count = str_count(mdgperson_txt$b3_repl,"r ")
mdgperson_txt$b6_split = strsplit(mdgperson_txt$b6,split=" ")
View(mdgperson_txt)
```

The new data frame as shown with the *View* command. We see that when a string is not found with the str_locate and stri_locate functions, N/A is returned (not 0):



	hh	state	urbrur	member	b3	b4	b5	b6	state_no	state_txt	urbrur_low	urbrur_up	b3_repl	state_urbrur	b3_find	b3_rfind	b6_count	b3_repl_count	b6_split
1	020074	02 Capital	Urban	1	Head	Male	39	Married - polygamy	02	Capital	urban	URBAN	Head	Capital Urban	2	2	1	0	c("Married", "-", "polygamy")
2	020074	02 Capital	Urban	2	Spouse	Female	21	Married - monogamy	02	Capital	urban	URBAN	Spouse	Capital Urban	6	6	1	0	c("Married", "-", "monogamy")
3	020074	02 Capital	Urban	3	Daughter/son	Male	16	Never married	02	Capital	urban	URBAN	Daughter or son	Capital Urban	7	7	3	2	c("Never", "married")
4	040024	04 East	Urban	1	Head	Male	37	Married - monogamy	04	East	urban	URBAN	Head	East Urban	2	2	1	0	c("Married", "-", "monogamy")
5	040024	04 East	Urban	2	Non relative	Female	23	Married - polygamy	04	East	urban	URBAN	Non relative	East Urban	6	12	1	0	c("Married", "-", "polygamy")
6	040024	04 East	Urban	3	Daughter/son	Female	17	Never married	04	East	urban	URBAN	Daughter or son	East Urban	7	7	3	2	c("Never", "married")
7	040024	04 East	Urban	4	Other relative	Female	9	Missing	04	East	urban	URBAN	Other relative	East Urban	4	14	0	1	Missing
8	050069	05 South	Rural	1	Head	Male	67	Married - monogamy	05	South	rural	RURAL	Head	South Rural	2	2	1	0	c("Married", "-", "monogamy")
9	050069	05 South	Rural	2	Spouse	Female	60	Married - monogamy	05	South	rural	RURAL	Spouse	South Rural	6	6	1	0	c("Married", "-", "monogamy")
10	050069	05 South	Rural	3	Grandchild	Female	16	Never married	05	South	rural	RURAL	Grandchild	South Rural	NA	NA	3	0	c("Never", "married")
11	060036	06 West	Urban	1	Head	Male	42	Married - monogamy	06	West	urban	URBAN	Head	West Urban	2	2	1	0	c("Married", "-", "monogamy")
12	060036	06 West	Urban	2	Spouse	Female	40	Married - monogamy	06	West	urban	URBAN	Spouse	West Urban	6	6	1	0	c("Married", "-", "monogamy")

When it comes to missing values, NA's, they are treated the same way as usual in R. This means that if an NA is part of our expression, the result will be NA. To avoid an NA as result we can use the na.rm = TRUE argument. Here is the same example as above, with R syntax:

```
test <- read.table(header=TRUE,text="
yearbirth income overtime
1974 234000     2320
1965 256000        0
1967 235350       NA
  NA 432330    33203
  NA      NA       NA
"
)
# Differences between sum and addition
test$income_total1 <- test$income+test$overtime
test$income_total2 <- rowSums(test[, c("income","overtime")],na.rm=TRUE)
test$age1 <- 2012-test$yearbirth
test$age2 <- 2012-rowSums(test['yearbirth'],na.rm=TRUE)
test
```

The *rowSums* function add sums for each selected row in our data frame. By using an empty string and a comma first, we select all rows. After the comma we name the variables to add together and also make sure to leave out NA's with the na.rm = TRUE argument. We are not allowed to subtract values with the rowSums function. Instead, we calculate the second age variable with the rowSums function even though it is only one variable, because then we can remove the NA's. The result is like this:

```
  yearbirth income overtime income_total1 income_total2 age1 age2
1      1974 234000     2320        236320        236320   38   38
2      1965 256000        0        256000        256000   47   47
3      1967 235350       NA            NA        235350   45   45
4        NA 432330    33203        465533        465533   NA 2012
5        NA     NA       NA            NA             0   NA 2012
```

If we don't want to repeat the data frame name every time we mention a variable, we can use the *with* function. First, we tell which data frame to use and then we do our calculation:

```
test$income_total1 <- with(test,income+overtime)
```

We can use the *dplyr* package to do the same calculations. Here we may use the sum function, which also take arguments with minus signs. First, we use the *rowwise* function to make sure the calculations are made for each row. Then we use the *mutate* function and create our new variables. Here we can also calculate the age2 variable:

```
test <- test %>%
  rowwise() %>%
  mutate(
    income_total1 = income+overtime,
    income_total2 = sum(income,overtime, na.rm = TRUE),
    age1 = 2012-yearbirth,
    age2 = sum(2012, -yearbirth, na.rm = TRUE)
    )
test
```

The result:

```
# A tibble: 5 x 7
  yearbirth income overtime income_total1 income_total2  age1  age2
      <int>  <int>    <int>         <int>         <int> <dbl> <dbl>
1      1974 234000     2320        236320        236320    38    38
2      1965 256000        0        256000        256000    47    47
3      1967 235350       NA            NA        235350    45    45
4        NA 432330    33203        465533        465533    NA  2012
5        NA     NA       NA            NA             0    NA  2012
```

18.5. Python

In python we have several functions and also many methods. The functions are not associated with any objects and can be invoked by its name. Methods are always associated with an object and cannot be invoked just by its name. For practical usage of functions and methods it is often just that we use different syntax to call functions compared to methods. Here is a small example where we define a function that adds to numbers together. The function is then called by its name:

```
def add(x, y):
    return x+y
add(1,4)
```

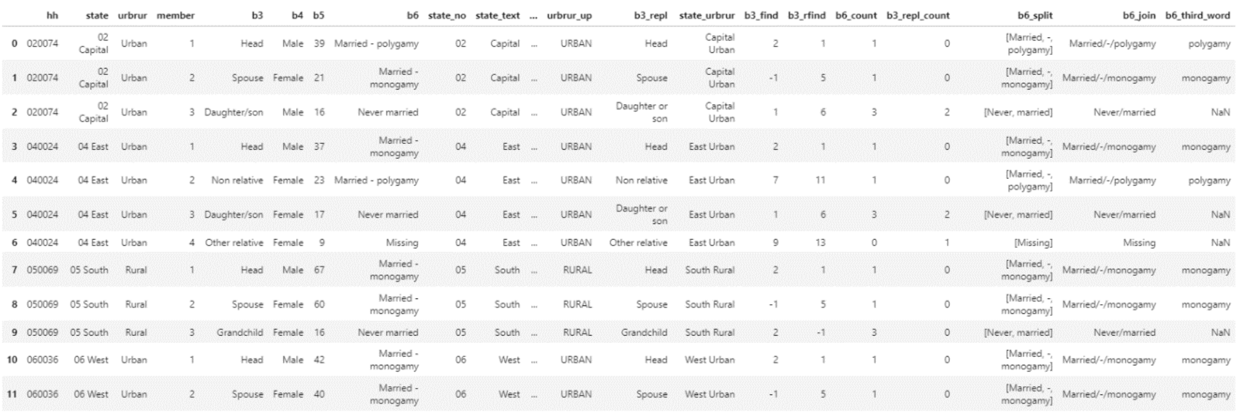
We can do the same with a method. Methods are always defined within an object which are defined by a class. Here we define a class object called arithmetic and within that class object we define two methods, addition and subtraction. The method is called both by the class object name and the method:

```
class arithmetic:
    def __init__(self, x, y):
                  self.x = x
                  self.y = y
    def addition(self):
        return self.x + self.y
    def subtraction(self):
        return self.x - self.y
arithmetic(1,4).addition()
```

Here are some examples on how we can use built-in functions for columns in data frames. We start with importing a data file with some string variables. Then we use different string functions to create new columns. Beware that for string functions, we add the str argument. For the substring there is no actual function, we just select the string with start and end positions (where 0 is the first position and the last position is not included in the string). When it comes to concatenation, we don't use a function, we just use the + sign. However, the columns must be of object (str) type, not category or numeric:

```
data="""
020074;02 Capital;Urban;1;Head;Male;39;Married - polygamy
020074;02 Capital;Urban;2;Spouse;Female;21;Married - monogamy
020074;02 Capital;Urban;3;Daughter/son;Male;16;Never married
040024;04 East;Urban;1;Head;Male;37;Married - monogamy
040024;04 East;Urban;2;Non relative;Female;23;Married - polygamy
040024;04 East;Urban;3;Daughter/son;Female;17;Never married
040024;04 East;Urban;4;Other relative;Female;9;Missing
050069;05 South;Rural;1;Head;Male;67;Married - monogamy
050069;05 South;Rural;2;Spouse;Female;60;Married - monogamy
050069;05 South;Rural;3;Grandchild;Female;16;Never married
060036;06 West;Urban;1;Head;Male;42;Married - monogamy
060036;06 West;Urban;2;Spouse;Female;40;Married - monogamy
"""
mdgperson_txt = pd.read_csv(
                StringIO(data),
                names=['hh', 'state', 'urbrur', 'member', 'b3', 'b4', 'b5',
'b6'],
                dtype={'hh': 'object', 'state': 'object', 'urbrur':
'category', 'b3': 'object', 'b4': 'object', 'b6': 'object'},
                header=None,
                sep=';'
                )
mdgperson_txt['state_no'] = mdgperson_txt['state'].str[0:2]
mdgperson_txt['state_text'] = mdgperson_txt['state'].str[3:]
mdgperson_txt['urbrur_low'] = mdgperson_txt['urbrur'].str.lower()
mdgperson_txt['urbrur_up'] = mdgperson_txt['urbrur'].str.upper()
mdgperson_txt['b3_repl'] = mdgperson_txt['b3'].str.replace('/',' or ')
mdgperson_txt['state_urbrur'] = mdgperson_txt['state'].str[3:] + ' ' +
mdgperson_txt['urbrur'].astype(str)
mdgperson_txt['b3_find'] = mdgperson_txt['b3'].str.find('a')
mdgperson_txt['b3_rfind'] = mdgperson_txt['b3'].str.rfind('e')
mdgperson_txt['b6_count'] = mdgperson_txt['b6'].str.count('e')
mdgperson_txt['b3_repl_count'] = mdgperson_txt['b3_repl'].str.count('r ')
mdgperson_txt['b6_split'] = mdgperson_txt['b6'].str.split(' ')
mdgperson_txt['b6_join'] = mdgperson_txt['b6_split'].str.join('/')
mdgperson_txt['b6_third_word'] = mdgperson_txt['b6_split'].str[2]
mdgperson_txt
```

For most of these string functions, we must insert *str*. before the actual function to make it work on data frames. We see that when a string is not found, the find and rfind functions return -1. For substrings, we use the slice technique where we define the slices (substring) within brackets ([]). When the column contains a list, the slice will be the elements in the list, not the positions within the whole text (compare the first and last of the examples above). Beware that the positions within the string starts with 0, not 1:

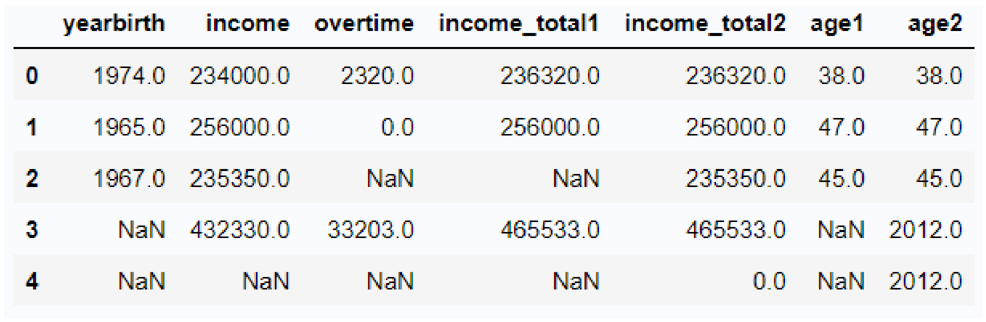


	hh	state	urbrur	member	b3	b4	b5	b6	state_no	state_text	...	urbrur_up	b3_repl	state_urbrur	b3_find	b3_rfind	b6_count	b3_repl_count	b6_split	b6_join	b6_third_word
0	020074	02 Capital	Urban	1	Head	Male	39	Married - polygamy	02	Capital	...	URBAN	Head	Capital Urban	2	1	1	0	[Married, -, polygamy]	Married/-/polygamy	polygamy
1	020074	02 Capital	Urban	2	Spouse	Female	21	Married - monogamy	02	Capital	...	URBAN	Spouse	Capital Urban	-1	5	1	0	[Married, -, monogamy]	Married/-/monogamy	monogamy
2	020074	02 Capital	Urban	3	Daughter/son	Male	16	Never married	02	Capital	...	URBAN	Daughter or son	Capital Urban	1	6	3	2	[Never, married]	Never/married	NaN
3	040024	04 East	Urban	1	Head	Male	37	Married - monogamy	04	East	...	URBAN	Head	East Urban	2	1	1	0	[Married, -, monogamy]	Married/-/monogamy	monogamy
4	040024	04 East	Urban	2	Non relative	Female	23	Married - polygamy	04	East	...	URBAN	Non relative	East Urban	7	11	1	0	[Married, -, polygamy]	Married/-/polygamy	polygamy
5	040024	04 East	Urban	3	Daughter/son	Female	17	Never married	04	East	...	URBAN	Daughter or son	East Urban	1	6	3	2	[Never, married]	Never/married	NaN
6	040024	04 East	Urban	4	Other relative	Female	9	Missing	04	East	...	URBAN	Other relative	East Urban	9	13	0	1	[Missing]	Missing	NaN
7	050069	05 South	Rural	1	Head	Male	67	Married - monogamy	05	South	...	RURAL	Head	South Rural	2	1	1	0	[Married, -, monogamy]	Married/-/monogamy	monogamy
8	050069	05 South	Rural	2	Spouse	Female	60	Married - monogamy	05	South	...	RURAL	Spouse	South Rural	-1	5	1	0	[Married, -, monogamy]	Married/-/monogamy	monogamy
9	050069	05 South	Rural	3	Grandchild	Female	16	Never married	05	South	...	RURAL	Grandchild	South Rural	2	-1	3	0	[Never, married]	Never/married	NaN
10	060036	06 West	Urban	1	Head	Male	42	Married - monogamy	06	West	...	URBAN	Head	West Urban	2	1	1	0	[Married, -, monogamy]	Married/-/monogamy	monogamy
11	060036	06 West	Urban	2	Spouse	Female	40	Married - monogamy	06	West	...	URBAN	Spouse	West Urban	-1	5	1	0	[Married, -, monogamy]	Married/-/monogamy	monogamy

When it comes to addition with NaN values, we can only use addition or subtraction. To deal with NaN's we can use the *fillna* function. With the *fillna* function we can treat the NaN values as zeros. Here is an example:

```
data="""
1974,234000,2320
1965,256000,0
1967,235350,.
.,432330,33203
.,.,.
"""
test = pd.read_csv(
        StringIO(data),
        names=['yearbirth', 'income', 'overtime'],
        dtype=float,
        na_values={'.', ' .'},
        header=None
       )

# Differences between sum and addition
test['income_total1'] = test['income'] + test['overtime']
test['income_total2'] = test['income'].fillna(0) + test['overtime'].fillna(0)
test['age1'] = 2012 - test['yearbirth']
test['age2'] = 2012 - test['yearbirth'].fillna(0)
test
```

This is the output:



	yearbirth	income	overtime	income_total1	income_total2	age1	age2
0	1974.0	234000.0	2320.0	236320.0	236320.0	38.0	38.0
1	1965.0	256000.0	0.0	256000.0	256000.0	47.0	47.0
2	1967.0	235350.0	NaN	NaN	235350.0	45.0	45.0
3	NaN	432330.0	33203.0	465533.0	465533.0	NaN	2012.0
4	NaN	NaN	NaN	NaN	0.0	NaN	2012.0

19. Missing values

As seen in the previous chapter missing values should be treated with care. Missing values are usually not included in calculations. They are also treated a little different in some situations across Sas, Spss, Stata, R and Python. We will show some of the differences in this chapter.

A string variable in Stata is missing when the value is an empty string. A blank (= one space) string is a valid value. In Sas both a space and an empty string are defined as missing values. Spss differs between system missing and user missing values. Only numeric variables have system missing values. User missing may be defined as any character for string variables and any number for numeric variables. However, we should be careful with what values we define as user missing and when we assign these values. If a numeric value is set as user missing in the data, we also have to define them as missing with the *Missing values* command. For string variables, it is usual to set missing to an empty string or a space. In Sas and Spss there is no difference between a space and an empty string when it comes to handling of missing values. If we set the user missing value to an empty string in Spss, a string of spaces will also be defined as user missing.

For numeric variables both Sas and Stata operates with several special missing values. The valid missing values are defined from .a to .z in Stata and .A to .Z in Sas. Otherwise, the default value of missing for numeric values in Sas and Stata is a dot (.). This makes it possible for us to define the dot as system missing and choose other values for user missing. Sas and Stata do not differ between system missing and user missing values when it comes to frequencies and tabulation the same way as Spss does.

In R there is only one missing value, and it is called NA (not available). However, there is also a value for impossible values (e.g., dividing by zero), NaN (not a number). The NA is the same for both character and numeric values.

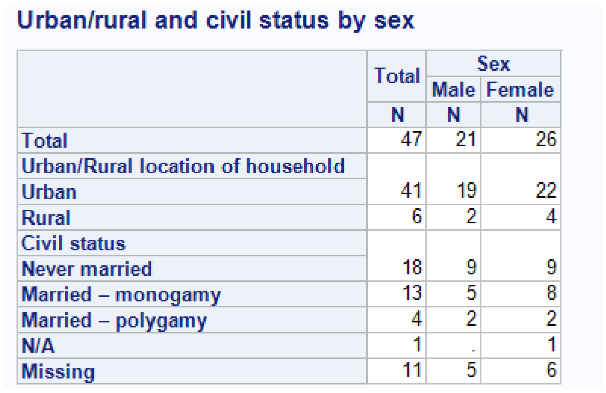
For Python, we have some different values for missing. The usual one is for numbers and is called NaN (not a number). We also have NaT (not a time, for time variables) and None which symbols and empty string value.

19.1. Sas

Here is a Data step where we set the Civil status (*b6*) to the special missing value .U when it miss a value where there should be one. We start with the *Data* statement and name the output dataset. Then we read the existing dataset with the *Set* statement. The *If* statement is used to conditionally set the value of *b6* to .U. Then we do the tabulation with *Proc tabulate*.

```
data mdgperson3;
 set mdg.mdgperson_nodup;
 if b5 >= 11 and b6 = . then b6 = .U;
run;
proc tabulate data=mdgperson3 missing f=11.;
 class urbrur b4 ;
 class b6 /preloadfmt order=data;
 table all='Total' urbrur='Urban/rural location of household' b6='Civil
Status'
       ,
       all='Total' b4='Sex';
 format urbrur urbrur. b6 civil_status. b4 sex.;
 title 'Urban/rural and civil status by sex';
run;
```

The output shows that both missing values are incorporated in the table:


Urban/rural and civil status by sex

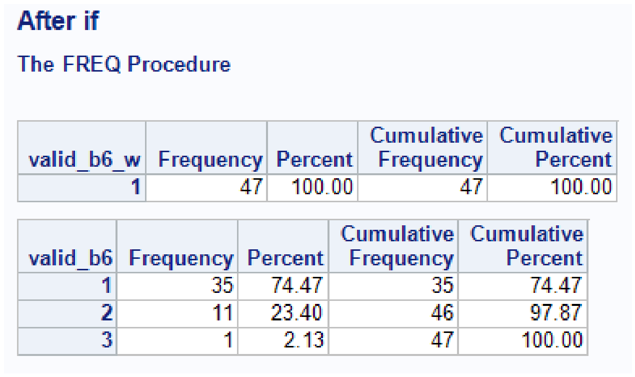
	Total	Sex	
		Male	Female
	N	N	N
Total	47	21	26
Urban/Rural location of household			
Urban	41	19	22
Rural	6	2	4
Civil status			
Never married	18	9	9
Married – monogamy	13	5	8
Married – polygamy	4	2	2
N/A	1	.	1
Missing	11	5	6

For Civil status the row with missing values should be excluded. There is no easy way to exclude them in Sas without excluding the observations from the whole table as well.

When we want to deal with missing values in an *If* construction, we must know that missing values are included in any condition which checks for values less than a number. For instance, the condition b6 < 7 will be true for all values less than 7, including all missing values. Hence the first *If* construction below will not give the expected result. Instead, we use the second *If* construction:

```
data mdgperson4;
 set mdgperson3;
 if (b6 <= 6) then
  valid_b6_w = 1;
 else if b6 = .U then
  valid_b6_w = 3;
 else if b6 = . then
  valid_b6_w = 2;
 else
  valid_b6_w = 9;

 if (1<= b6 <= 6) then
  valid_b6 = 1;
 else if b6 = .U then
  valid_b6 = 3;
 else if b6 = . then
  valid_b6 = 2;
 else
  valid_b6 = 9;
run;

proc freq data=mdgperson4;
 tables valid_b6_w valid_b6/missing;
 title 'After if';
run;
```

With the frequencies made with the *Proc freq* above, we see the differences between the two *If* constructions:



After if

The FREQ Procedure

valid_b6_w	Frequency	Percent	Cumulative Frequency	Cumulative Percent
1	47	100.00	47	100.00

valid_b6	Frequency	Percent	Cumulative Frequency	Cumulative Percent
1	35	74.47	35	74.47
2	11	23.40	46	97.87
3	1	2.13	47	100.00

19.2. Spss

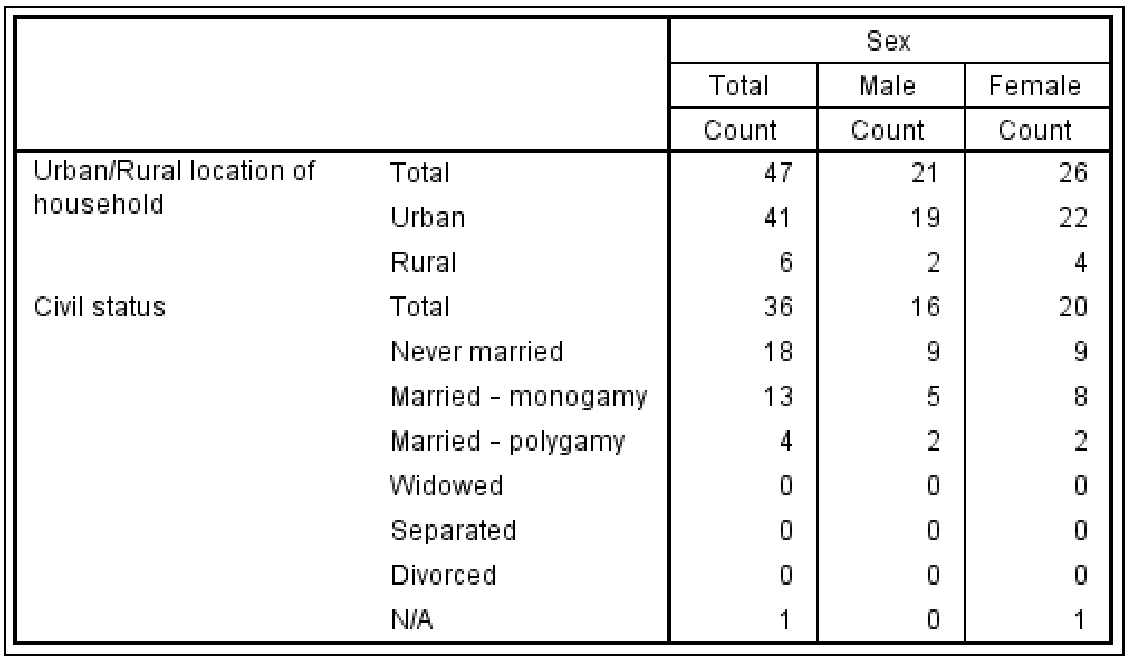
For numeric variables in Spss we have to choose a number as user missing value. For categorical variables we usually set user missing to a higher number than any of the valid values. When the value is set, we must remember to use the *Missing values* command to define the value as user missing. For scale variables we usually do not differ between system and user missing values. A normal approach for categorical variables is to set them to system missing when they are not supposed to have a value. This is typical for questions in a survey which have been skipped due to answers to a previous question. Those who are not to answer the question will be given system missing for these variables. User missing is set when there should be a value, but none is entered or imputed. The categorical user missing values should be included in tabulations, but not the system missing values.

We want to create a table with urban/rural and civil status in the rows and sex in the columns. For the question of civil status, it is only asked persons 11 years and above. Hence, we want to set missing values for people with age 11 and above to user missing and include them in the table. People below 11 should not be included in this category. First, we use an *If* command to set values to user missing. Then we use the *Missing values* command to define user missing values. Finally, we use the *Ctables* command to create the table:

```
GET FILE='mdgperson_nodup.sav'.

IF (b5 >= 11 and missing(b6)  = 1) b6 = 9.

MISSING VALUES b6 (9).

CTABLES
  /VLABELS VARIABLES=b6 b4 DISPLAY=LABEL
  /TABLE urbrur + b6 BY b4 [COUNT F40.0]
  /CATEGORIES VARIABLES=urbrur ORDER=A KEY=VALUE EMPTY=INCLUDE TOTAL=YES
POSITION=BEFORE
  /CATEGORIES VARIABLES=b6 ORDER=A KEY=VALUE EMPTY=INCLUDE TOTAL=YES
POSITION=BEFORE MISSING=INCLUDE
  /CATEGORIES VARIABLES=b4 ORDER=A KEY=VALUE EMPTY=INCLUDE TOTAL=YES
POSITION=BEFORE.
```

We see that the table includes user missing (N/A), but not system missing for the Civil status category:

```
h:\mdg\data\mdgperson_nodup.sav
```



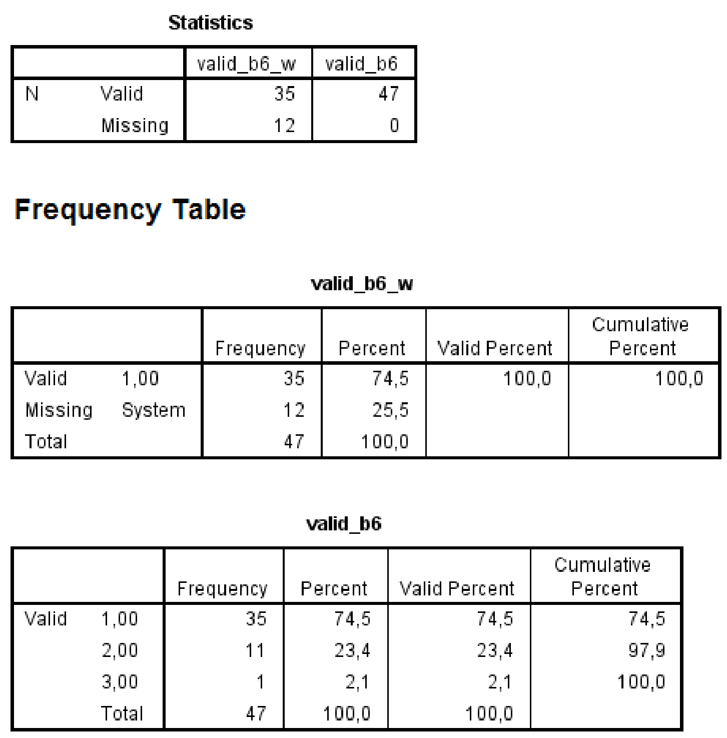
		Sex		
		Total	Male	Female
		Count	Count	Count
Urban/Rural location of household	Total	47	21	26
	Urban	41	19	22
	Rural	6	2	4
Civil status	Total	36	16	20
	Never married	18	9	9
	Married - monogamy	13	5	8
	Married - polygamy	4	2	2
	Widowed	0	0	0
	Separated	0	0	0
	Divorced	0	0	0
	N/A	1	0	1

We have to be aware how Spss deals with missing when it comes to conditions. If we want to check against a missing value with a *Do if* command, we must use the functions *Missing* or *Sysmis*, otherwise our condition will never be true for missing values. In addition, we must start with our check for missing values.

This is a syntax example which shows two different *Do if* constructions, only the last one will give the intended result:

```
DO IF (b6 <= 6).
 COMPUTE valid_b6_w = 1.
ELSE IF (b6 = 9).
  COMPUTE valid_b6_w = 3.
ELSE.
 COMPUTE valid_b6_w = 2.
END IF.

DO IF (sysmis(b6) = 1).
  COMPUTE valid_b6 = 2.
ELSE IF (missing(b6) = 1).
  COMPUTE valid_b6 = 3.
ELSE IF (b6 <6 ).
  COMPUTE valid_b6 = 1.
ELSE.
  COMPUTE valid_b6 = 9.
END IF.

FREQUENCIES valid_b6_w valid_b6.
```

As our frequency table shows, the variable *valid_b6_w* is not given a value when *b6* is missing:



Statistics

		valid_b6_w	valid_b6
N	Valid	35	47
	Missing	12	0

Frequency Table

valid_b6_w

		Frequency	Percent	Valid Percent	Cumulative Percent
Valid	1,00	35	74,5	100,0	100,0
Missing	System	12	25,5		
Total		47	100,0		

valid_b6

		Frequency	Percent	Valid Percent	Cumulative Percent
Valid	1,00	35	74,5	74,5	74,5
	2,00	11	23,4	23,4	97,9
	3,00	1	2,1	2,1	100,0
	Total	47	100,0	100,0	

19.3. Stata

As Sas, Stata uses numbers as missing values. Where missing values are lower than the lowest legal number in Sas it is higher than the highest legal in Stata. This means that missing values will be included in all conditions where we use > or >=. When we use conditions like these and don't want to include the missing values, we have to add another condition to exclude them.

When we want to set a special missing value for the civil status when it is missing, and age is above 10 we can do like this:

```
use "mdgperson_nodup.dta", clear
replace b6 = .u if b5 >= 11 & b6 == .
```

In the tabulation the missing value . is omitted, but the user defined missing .u is included. Here is syntax to create two tables:

```
table urbrur b4, scol col row
table b6 b4, scol col row
```

The tabulation omits the observations with system user missing values. That is why the total is different in the tables. User missing is as we see included in the table:

```
. table urbrur b4,  scol col row

----------------------------------
Urban/Rur |
al        |
location  |
of        |           Sex
household |   Male  Female   Total
----------+-----------------------
    Urban |     19      22      41
    Rural |      2       4       6
          |
    Total |     21      26      47
----------------------------------
```

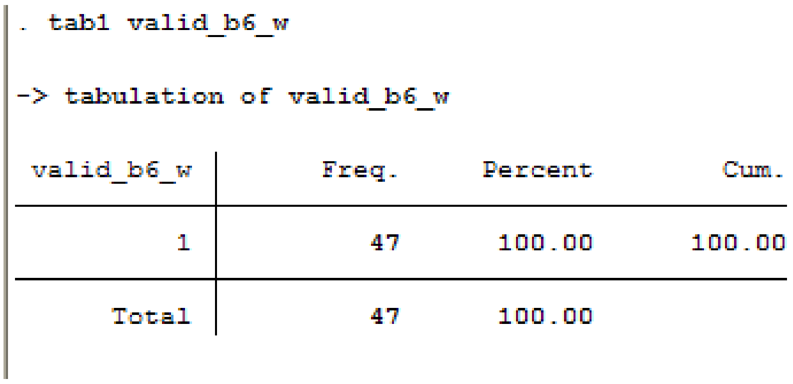
```
. table b6 b4,  scol col row

-------------------------------------------
                  |          Sex
     Civil status |   Male  Female    Total
------------------+------------------------
    Never married |      9       9       18
Married - monogamy|      5       8       13
Married - polygamy|      2       2        4
              N/A |              1        1
                  |
            Total |     16      20       36
-------------------------------------------
```

It does not seem to be an easy way to concatenate the row variables into one table in Stata, hence we create two separate tables.

Stata does not have the same *If* construction as Sas and Spss for normal use. Instead, we use either a combination of *generate* and *replace* commands or a *recode* command. We may do it in the wrong way when it comes to the missing values:

```
generate valid_b6_w = 3 if b6 == .u
replace  valid_b6_w = 2 if b6 == .
replace  valid_b6_w = 9 if b6 > 6 & b6 < .
replace  valid_b6_w = 1 if b6 >= 1

tab1 valid_b6_w
```

The order of the commands above is wrong as the condition in the last replace includes the missing values:

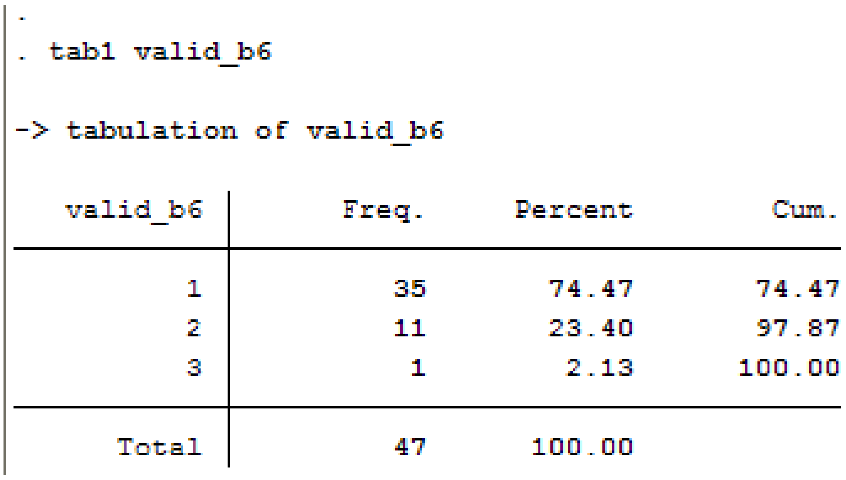


```
. tab1 valid_b6_w

-> tabulation of valid_b6_w

 valid_b6_w |      Freq.     Percent        Cum.
------------+-----------------------------------
          1 |         47      100.00      100.00
------------+-----------------------------------
      Total |         47      100.00
```

Instead, we change the order of the commands:

```
use "mdgperson_nodup.dta", clear
replace b6 = .u if b5 >= 11 & b6 == .

generate valid_b6 = 1 if b6 >= 1
replace  valid_b6 = 3 if b6 == .u
replace  valid_b6 = 2 if b6 == .
replace  valid_b6 = 9 if b6 > 6 & b6 < .
tab1 valid_b6
```

Now the recode is as intended:



```
.
. tab1 valid_b6

-> tabulation of valid_b6

   valid_b6 |      Freq.     Percent        Cum.
------------+-----------------------------------
          1 |         35       74.47       74.47
          2 |         11       23.40       97.87
          3 |          1        2.13      100.00
------------+-----------------------------------
      Total |         47      100.00
```

For a recode like the one above it is usually better to use the *recode* command:

```
recode b6 (1/6 = 1) (.u = 3) (. = 2) (else = 9), gen(valid_b6)
```

More about the *recode* command is found in the chapter Recoding, page 118.

19.4.R

When we deal with missing values in R, it is useful to know about some functions.

• is.na	Check if a value is NA or not
• na.rm	Remove (TRUE) or keep (FALSE) NA's
• complete.cases	Check if a row has NA's

Some examples:

```
> mdgperson$b5
 [1] 10 39 20 20 33 23 16 24 16 60 27  8  3  8 18 14 16 13 21  5 13 18  9 41
10
[26] 31 67 17 NA 22 21 16  8  7 19 42 45 20 20  9 11 30 40 17  1 12 37 17
> is.na(mdgperson$b5)
 [1] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
[13] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
[25] FALSE FALSE FALSE FALSE  TRUE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
[37] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
> mean(mdgperson$b5)
[1] NA
> mean(mdgperson$b5,na.rm = TRUE)
[1] 21.14894
> complete.cases(mdgperson)
 [1] FALSE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE FALSE
[13] FALSE FALSE  TRUE  TRUE  TRUE  TRUE  TRUE FALSE  TRUE  TRUE FALSE  TRUE
[25] FALSE  TRUE  TRUE  TRUE FALSE  TRUE  TRUE  TRUE FALSE FALSE  TRUE  TRUE
[37]  TRUE  TRUE  TRUE FALSE FALSE  TRUE  TRUE  TRUE FALSE  TRUE  TRUE  TRUE
```

We see that numeric operations like mean are not calculated if one or more values are NA unless we add the *na.rm* = *TRUE* argument.

We can create a variable valid_b6 that is 1 when b6 has a value and NA if it is missing and make a frequency table for the new variable:

```
mdgperson_nodup$valid_b6 <- ifelse(is.na(mdgperson_nodup$b6),NA,1)
as.data.frame(addmargins(table(valid_b6=mdgperson_nodup$valid_b6,exclude = NU
LL)))
```

Here is the frequency table:

```
  valid_b6 Freq
1        1   35
2     <NA>   12
3      Sum   47
```

19.5. Python

We should always be aware of missing values and how we can deal with them. Python operates with both NaN and None as missing values. There are some differences between NaN and None. A comparison with NaN is never true, but a comparison with None can be true. This small program shows that:

```
print('None == None is', None == None)
print('np.nan == np.nan is', np.nan == np.nan)
```

The printed output:

```
None == None is True
np.nan == np.nan is False
```

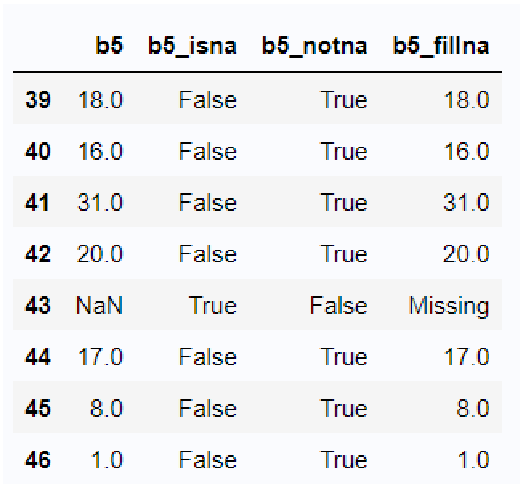
To deal with missing values we have some functions:

- isna   Check if value is NaN (can also use the alias isnull). Returns True or False.
- notna   Check if value is not NaN (can also use the alias notnull). Returns True or False.
- dropna   Drop row if NaN appears in at least one element specified
- fillna   Replace NaN with specified value

Here is an example where we create a data frame, *p*, with the *b5* column and add new columns with the *isna, notna* and *fillna* functions:

```
p = pd.DataFrame(persons['b5'])
p['b5_isna'] = persons['b5'].isna()
p['b5_notna'] = persons['b5'].notna()
p['b5_fillna'] = persons['b5'].fillna('Missing')
p.tail(8)
```

The list of the last 8 rows:



	b5	b5_isna	b5_notna	b5_fillna
39	18.0	False	True	18.0
40	16.0	False	True	16.0
41	31.0	False	True	31.0
42	20.0	False	True	20.0
43	NaN	True	False	Missing
44	17.0	False	True	17.0
45	8.0	False	True	8.0
46	1.0	False	True	1.0

We can drop rows with the *dropna* function. Either we can drop rows where at least one of the columns are missing or we can drop rows based on a subset of columns. Here are examples on both:

```
persons.dropna()
persons.dropna(subset=['b5'])
```

By default, calculations are based on non-missing values. We can calculate the count, mean, and sum for a data frame and list them:

```
persons['b5'].count(), persons['b5'].sum(), persons['b5'].mean()
```

The list:

```
(46, 977.0, 21.23913043478261)
```

20. Date and time formats

We operate with three different variable types for date and time:

• Date	A date, e.g., March 9, 1954
• Time	A time, e.g., 14:13:22.00
• Datetime	A time within a date, e.g., March 9, 1954 14:13:22.00

There are some issues to be aware of when we are working with time and date variables. A time variable consists of both a date and a time. A date variable is limited to contain the value of a specific date. If we define times and dates as time and date variables, we are able to use lots of built-in facilities in the software. A date variable is stored as a number in a row. Sas and Stata have set January 1, 1960 as date 0, R starts at January 1, 1970 for R. All previous dates have negative values and newer dates are positive. Spss starts with the number 86400 at the beginning of October 14, 1582 and adds 1 for each second since then. It does not differ between time and date variables when it comes to how they are stored. To extract a date from a time variable we use the *Xdate.date* function (see page 127).

Sas, Stata and R differ between datetime, date and time variables. A datetime variable in Sas counts the seconds as a number starting at the beginning of January 1, 1960 as second 0. Milliseconds are stored as decimals to these numbers. A time variable has a value from 0 to 86400, which is the number of seconds in a day. Stata just count milliseconds from the beginning of January 1, 1960, as being millisecond 0. When we convert data between these softwares, we may have to recalculate time and date values to make sure they are correct. Invalid dates are set to missing values when the data is imported. R use January 1, 1970 as their date 0. However, in R we can define the 0 date when we convert date and datetime from other formats.

There are different formats we use to show the time and date variables. Once a variable is stored as date or time, we can use these formats. We can also use functions to extract parts of a time or date, for instance to extract the year. Other functions are used to calculate time spans between two points in time.

See more about time and date variables in the next chapter.

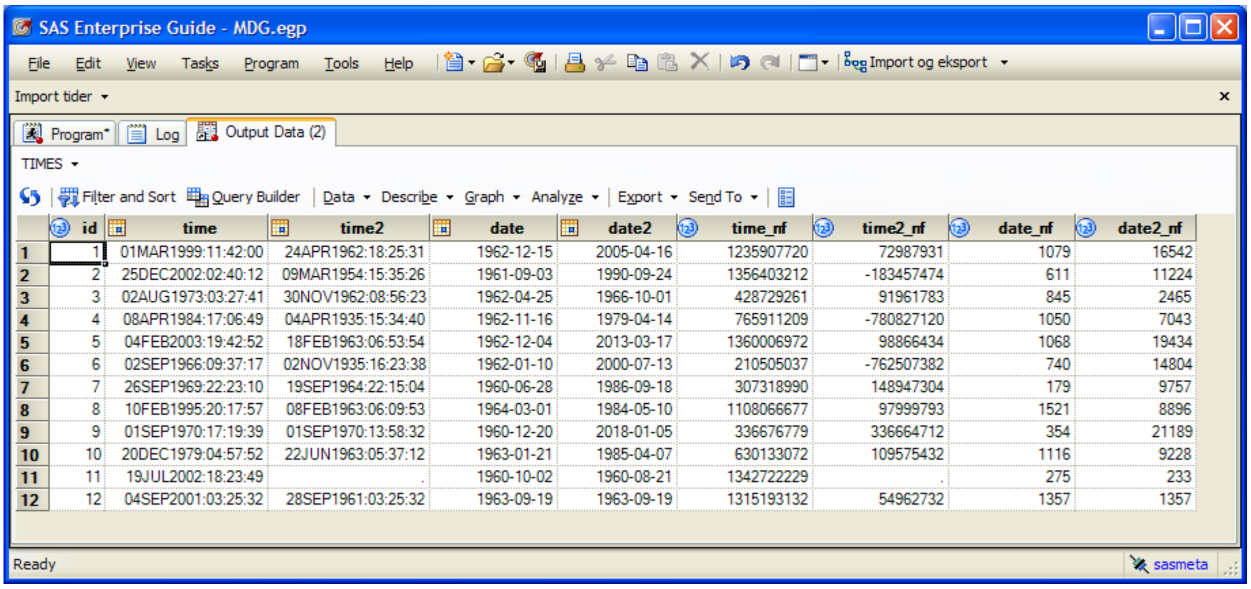
20.1. Sas

We want to read a data file with date and time variables to a Sas dataset. To make sure the variables are stored as times or dates, we use Sas informats when we read the data. To show the variables in a readable way we use the *Format* statement to format the variables. Here we also make unformatted copies of the variables (with suffix _nf) to show which values are actually stored for times and dates on the dataset:

```
data times;
 infile cards dlm=',' dsd truncover;
 input id time :  datetime19. time2 :  datetime19. date : yymmdd10. date2 :
yymmdd10. ;
 time_nf = time;
 time2_nf = time2;
 date_nf = date;
 date2_nf = date2;
 format time time2 datetime19. date date2 yymmdd10.;
cards;
01,01-MAR-1999 11:42:00,24-APR-1962 18:25:31,1962-12-15,2005-04-16
02,25-DEC-2002 02:40:12,09-MAR-1954 15:35:26,1961-09-03,1990-09-24
```

```
03,02-AUG-1973 03:27:41,30-NOV-1962 08:56:23,1962-04-25,1966-10-01
04,08-APR-1984 17:06:49,04-APR-1935 15:34:40,1962-11-16,1979-04-14
05,04-FEB-2003 19:42:52,18-FEB-1963 06:53:54,1962-12-04,2013-03-17
06,02-SEP-1966 09:37:17,02-NOV-1935 16:23:38,1962-01-10,2000-07-13
07,26-SEP-1969 22:23:10,19-SEP-1964 22:15:04,1960-06-28,1986-09-18
08,10-FEB-1995 20:17:57,08-FEB-1963 06:09:53,1964-03-01,1984-05-10
09,01-SEP-1970 17:19:39,01-SEP-1970 13:58:32,1960-12-20,2018-01-05
10,20-DEC-1979 04:57:52,22-JUN-1963 05:37:12,1963-01-21,1985-04-07
11,19-JUL-2002 18:23:49,29-FEB-1961 15:41:51,1960-10-02,1960-08-21
12,04-SEP-2001 03:25:32,28-SEP-1961 03:25:32,1963-09-19,1963-09-19
;
run;
```

When the data is imported to Sas, and new variables and formats added, the dataset looks like this:


SAS Enterprise Guide - MDG.egp
File Edit View Tasks Program Tools Help
Import og eksport
Import tider
Program* Log Output Data (2)
TIMES
Filter and Sort Query Builder Data Describe Graph Analyze Export Send To
Ready
sasmeta

	id	time	time2	date	date2	time_nf	time2_nf	date_nf	date2_nf
1	1	01MAR1999:11:42:00	24APR1962:18:25:31	1962-12-15	2005-04-16	1235907720	72987931	1079	16542
2	2	25DEC2002:02:40:12	09MAR1954:15:35:26	1961-09-03	1990-09-24	1356403212	-183457474	611	11224
3	3	02AUG1973:03:27:41	30NOV1962:08:56:23	1962-04-25	1966-10-01	428729261	91961783	845	2465
4	4	08APR1984:17:06:49	04APR1935:15:34:40	1962-11-16	1979-04-14	765911209	-780827120	1050	7043
5	5	04FEB2003:19:42:52	18FEB1963:06:53:54	1962-12-04	2013-03-17	1360006972	98866434	1068	19434
6	6	02SEP1966:09:37:17	02NOV1935:16:23:38	1962-01-10	2000-07-13	210505037	-762507382	740	14804
7	7	26SEP1969:22:23:10	19SEP1964:22:15:04	1960-06-28	1986-09-18	307318990	148947304	179	9757
8	8	10FEB1995:20:17:57	08FEB1963:06:09:53	1964-03-01	1984-05-10	1108066677	97999793	1521	8896
9	9	01SEP1970:17:19:39	01SEP1970:13:58:32	1960-12-20	2018-01-05	336676779	336664712	354	21189
10	10	20DEC1979:04:57:52	22JUN1963:05:37:12	1963-01-21	1985-04-07	630133072	109575432	1116	9228
11	11	19JUL2002:18:23:49	.	1960-10-02	1960-08-21	1342722229	.	275	233
12	12	04SEP2001:03:25:32	28SEP1961:03:25:32	1963-09-19	1963-09-19	1315193132	54962732	1357	1357

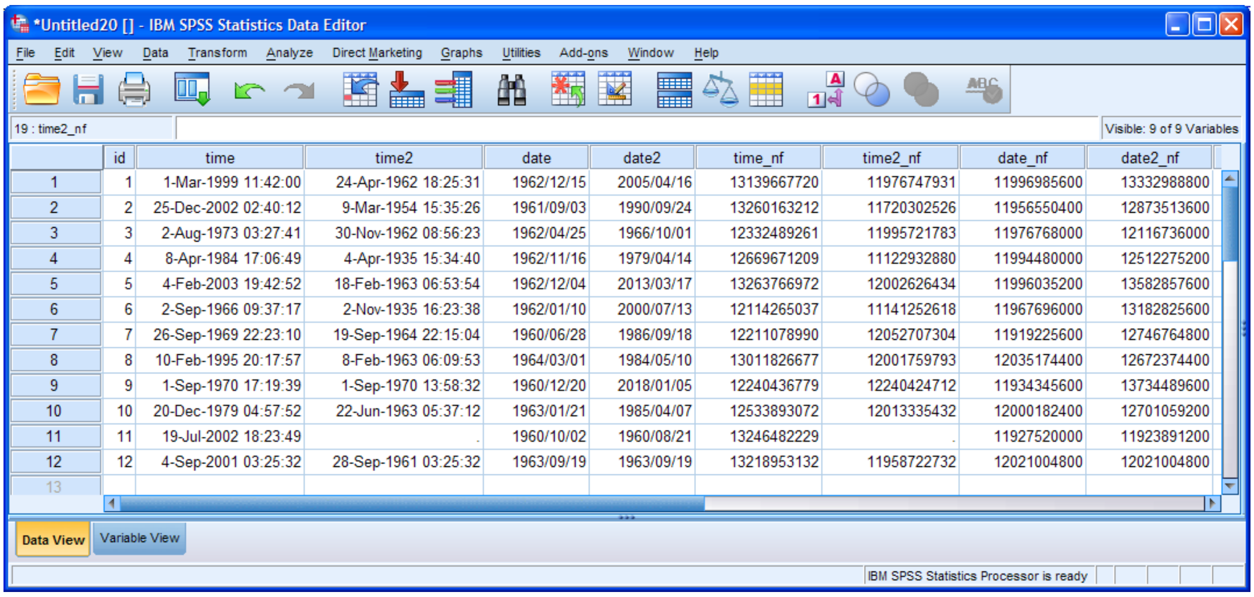
20.2. Spss

In Spss we can also use formatting when we import the data file. Here we use the *datetime* format for the time variables and the *sdate* format for the date variables. When the data is imported, we add new variables with the *Compute* commands. Finally, we add formats with the *Formats* command and save the dataset with the *Save* command.

```
DATA LIST /
 id    1-2   (f)
 time  4-23  (datetime)
 time2 25-44 (datetime)
 date  46-55 (sdate)
 date2 57-66 (sdate) .
BEGIN DATA
01,01-MAR-1999 11:42:00,24-APR-1962 18:25:31,1962-12-15,2005-04-16
02,25-DEC-2002 02:40:12,09-MAR-1954 15:35:26,1961-09-03,1990-09-24
03,02-AUG-1973 03:27:41,30-NOV-1962 08:56:23,1962-04-25,1966-10-01
04,08-APR-1984 17:06:49,04-APR-1935 15:34:40,1962-11-16,1979-04-14
05,04-FEB-2003 19:42:52,18-FEB-1963 06:53:54,1962-12-04,2013-03-17
06,02-SEP-1966 09:37:17,02-NOV-1935 16:23:38,1962-01-10,2000-07-13
07,26-SEP-1969 22:23:10,19-SEP-1964 22:15:04,1960-06-28,1986-09-18
08,10-FEB-1995 20:17:57,08-FEB-1963 06:09:53,1964-03-01,1984-05-10
09,01-SEP-1970 17:19:39,01-SEP-1970 13:58:32,1960-12-20,2018-01-05
10,20-DEC-1979 04:57:52,22-JUN-1963 05:37:12,1963-01-21,1985-04-07
11,19-JUL-2002 18:23:49,29-FEB-1961 15:41:51,1960-10-02,1960-08-21
12,04-SEP-2001 03:25:32,28-SEP-1961 03:25:32,1963-09-19,1963-09-19
```

```
END DATA.
COMPUTE time_nf = time.
COMPUTE time2_nf = time2.
COMPUTE date_nf = date.
COMPUTE date2_nf = date2.
FORMATS time_nf time2_nf date_nf date2_nf (f16).
EXECUTE.
SAVE OUTFILE='times.sav'.
```

We see in the imported dataset that both the time and date values are stored as time variables. For the date variables the time will be set to 00:00:00:


*Untitled20 [] - IBM SPSS Statistics Data Editor

19 : time2_nf

Visible: 9 of 9 Variables

	id	time	time2	date	date2	time_nf	time2_nf	date_nf	date2_nf
1	1	1-Mar-1999 11:42:00	24-Apr-1962 18:25:31	1962/12/15	2005/04/16	13139667720	11976747931	11996985600	13332988800
2	2	25-Dec-2002 02:40:12	9-Mar-1954 15:35:26	1961/09/03	1990/09/24	13260163212	11720302526	11956550400	12873513600
3	3	2-Aug-1973 03:27:41	30-Nov-1962 08:56:23	1962/04/25	1966/10/01	12332489261	11995721783	11976768000	12116736000
4	4	8-Apr-1984 17:06:49	4-Apr-1935 15:34:40	1962/11/16	1979/04/14	12669671209	11122932880	11994480000	12512275200
5	5	4-Feb-2003 19:42:52	18-Feb-1963 06:53:54	1962/12/04	2013/03/17	13263766972	12002626434	11996035200	13582857600
6	6	2-Sep-1966 09:37:17	2-Nov-1935 16:23:38	1962/01/10	2000/07/13	12114265037	11141252618	11967696000	13182825600
7	7	26-Sep-1969 22:23:10	19-Sep-1964 22:15:04	1960/06/28	1986/09/18	12211078990	12052707304	11919225600	12746764800
8	8	10-Feb-1995 20:17:57	8-Feb-1963 06:09:53	1964/03/01	1984/05/10	13011826677	12001759793	12035174400	12672374400
9	9	1-Sep-1970 17:19:39	1-Sep-1970 13:58:32	1960/12/20	2018/01/05	12240436779	12240424712	11934345600	13734489600
10	10	20-Dec-1979 04:57:52	22-Jun-1963 05:37:12	1963/01/21	1985/04/07	12533893072	12013335432	12000182400	12701059200
11	11	19-Jul-2002 18:23:49	.	1960/10/02	1960/08/21	13246482229	.	11927520000	11923891200
12	12	4-Sep-2001 03:25:32	28-Sep-1961 03:25:32	1963/09/19	1963/09/19	13218953132	11958722732	12021004800	12021004800
13									

IBM SPSS Statistics Processor is ready

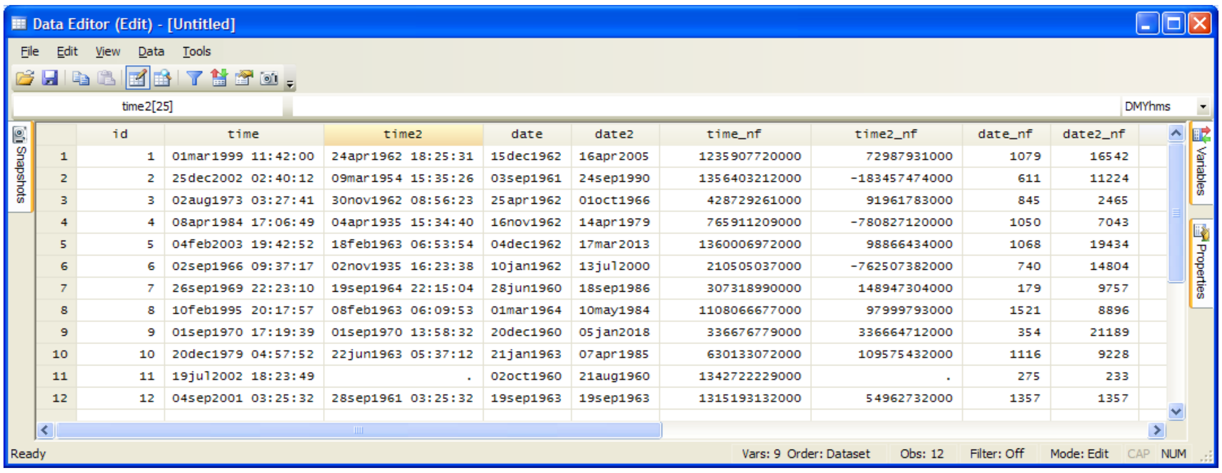
20.3. Stata

In Stata embedded files have limited functionality when it comes to date and time formats. We start with importing the variables as strings. The time variables should be defined as numeric double for the precision of the times. For the time variables we then concatenate the day with the time and convert to a time variable with the *clock* function. Finally, we format the variables and delete the temporary variables. For the date variables we convert them with the *date* function and add formats.

```
clear
input id str11 day str8 hour str11 day2 str8 hour2 str10 datec str10 date2c
01 01-MAR-1999 11-42-00 24-APR-1962 18-25-31 1962-12-15 2005-04-16
02 25-DEC-2002 02-40-12 09-MAR-1954 15-35-26 1961-09-03 1990-09-24
03 02-AUG-1973 03-27-41 30-NOV-1962 08-56-23 1962-04-25 1966-10-01
04 08-APR-1984 17-06-49 04-APR-1935 15-34-40 1962-11-16 1979-04-14
05 04-FEB-2003 19-42-52 18-FEB-1963 06-53-54 1962-12-04 2013-03-17
06 02-SEP-1966 09-37-17 02-NOV-1935 16-23-38 1962-01-10 2000-07-13
07 26-SEP-1969 22-23-10 19-SEP-1964 22-15-04 1960-06-28 1986-09-18
08 10-FEB-1995 20-17-57 08-FEB-1963 06-09-53 1964-03-01 1984-05-10
09 01-SEP-1970 17-19-39 01-SEP-1970 13-58-32 1960-12-20 2018-01-05
10 20-DEC-1979 04-57-52 22-JUN-1963 05-37-12 1963-01-21 1985-04-07
11 19-JUL-2002 18-23-49 29-FEB-1961 15-41-51 1960-10-02 1960-08-21
12 04-SEP-2001 03-25-32 28-SEP-1961 03-25-32 1963-09-19 1963-09-19
end

egen timec = concat(day hour) ,punct(" ")
gen double time = clock(timec,"DMYhms")
egen time2c = concat(day2 hour2) ,punct(" ")
```

```
gen double time2 = clock(time2c,"DMYhms")
gen date = date(datec,"YMD")
gen date2 = date(date2c,"YMD")
gen double time_nf = time
gen double time2_nf = time2
gen date_nf = date
gen date2_nf = date2
format time time2 %tc
format date date2 %td
format time_nf time2_nf %17.0f
drop day day2 hour hour2 timec time2c datec date2c
save "times.dta"
```

The time variables are defined in milliseconds starting at the beginning of January 1, 1960:



	id	time	time2	date	date2	time_nf	time2_nf	date_nf	date2_nf
1	1	01mar1999 11:42:00	24apr1962 18:25:31	15dec1962	16apr2005	1235907720000	72987931000	1079	16542
2	2	25dec2002 02:40:12	09mar1954 15:35:26	03sep1961	24sep1990	1356403212000	-183457474000	611	11224
3	3	02aug1973 03:27:41	30nov1962 08:56:23	25apr1962	01oct1966	428729261000	91961783000	845	2465
4	4	08apr1984 17:06:49	04apr1935 15:34:40	16nov1962	14apr1979	765911209000	-780827120000	1050	7043
5	5	04feb2003 19:42:52	18feb1963 06:53:54	04dec1962	17mar2013	1360006972000	98866434000	1068	19434
6	6	02sep1966 09:37:17	02nov1935 16:23:38	10jan1962	13jul2000	210505037000	-762507382000	740	14804
7	7	26sep1969 22:23:10	19sep1964 22:15:04	28jun1960	18sep1986	307318990000	148947304000	179	9757
8	8	10feb1995 20:17:57	08feb1963 06:09:53	01mar1964	10may1984	1108066677000	97999793000	1521	8896
9	9	01sep1970 17:19:39	01sep1970 13:58:32	20dec1960	05jan2018	336676779000	336664712000	354	21189
10	10	20dec1979 04:57:52	22jun1963 05:37:12	21jan1963	07apr1985	630133072000	109575432000	1116	9228
11	11	19jul2002 18:23:49	.	02oct1960	21aug1960	1342722229000	.	275	233
12	12	04sep2001 03:25:32	28sep1961 03:25:32	19sep1963	19sep1963	1315193132000	54962732000	1357	1357

Ready Vars: 9 Order: Dataset Obs: 12 Filter: Off Mode: Edit CAP NUM

20.4.R

R is good at date and time processing. We can import most different date and time formats and store them as date, datetime or time variables. Here is the same data as used previously, now imported to a R data frame:

```
times <- read.csv(sep=",",header=FALSE,
         col.names=c("id","time","time2","date", "date2"),
         colClasses=c("character","character","character","Date","Date"),
text="
01,01-MAR-1999 11:42:00,24-APR-1962 18:25:31,1962-12-15,2005-04-16
02,25-DEC-2002 02:40:12,09-MAR-1954 15:35:26,1961-09-03,1990-09-24
03,02-AUG-1973 03:27:41,30-NOV-1962 08:56:23,1962-04-25,1966-10-01
04,08-APR-1984 17:06:49,04-APR-1935 15:34:40,1962-11-16,1979-04-14
05,04-FEB-2003 19:42:52,18-FEB-1963 06:53:54,1962-12-04,2013-03-17
06,02-SEP-1966 09:37:17,02-NOV-1935 16:23:38,1962-01-10,2000-07-13
07,26-SEP-1969 22:23:10,19-SEP-1964 22:15:04,1960-06-28,1986-09-18
08,10-FEB-1995 20:17:57,08-FEB-1963 06:09:53,1964-03-01,1984-05-10
09,01-SEP-1970 17:19:39,01-SEP-1970 13:58:32,1960-12-20,2018-01-05
10,20-DEC-1979 04:57:52,22-JUN-1963 05:37:12,1963-01-21,1985-04-07
11,19-JUL-2002 18:23:49,29-FEB-1961 15:41:51,1960-10-02,1960-08-21
12,04-SEP-2001 03:25:32,28-SEP-1961 03:25:32,1963-09-19,1963-09-19
")
```

For dates we can define date formats directly in the import, but for datetime variables we first define them as characters. After they are imported, we can convert into real datetime variables.

The structure R data frame after import:

```
str(times)
```

```
'data.frame':     12 obs. of  5 variables:

 $ id   : chr  "01" "02" "03" "04" ...

 $ time : chr  "01-MAR-1999 11:42:00" "25-DEC-2002 02:40:12" "02-AUG-1973 03:
27:41" "08-APR-1984 17:06:49" ...

 $ time2: chr  "24-APR-1962 18:25:31" "09-MAR-1954 15:35:26" "30-NOV-1962 08:
56:23" "04-APR-1935 15:34:40" ...

 $ date : Date, format: "1962-12-15" "1961-09-03" "1962-04-25" "1962-11-16" .
..

 $ date2: Date, format: "2005-04-16" "1990-09-24" "1966-10-01" "1979-04-14" .
..
```

Even though the datetime variables look fine, they are not real datetime variables, but character variables. To be able to use calculations on them and to format them, we have to convert them to real datetime variables. We use the as.POSIXct function for this:

```
times$time  <- as.POSIXct(times$time,format="%d-%b-%Y %H:%M:%S")
times$time2  <- as.POSIXct(times$time2,format="%d-%b-%Y %H:%M:%S")
```

The datetime format is given by these parameters:

%a	Abbreviated weekday
%A	Full weekday
%b	Abbreviated month
%B	Full month
%c	Locale-specific date and time
%d	Decimal date
%H	Decimal hours (24 hour)
%I	Decimal hours (12 hour)
%j	Decimal day of the year
%m	Decimal month
%M	Decimal minute
%p	Locale-specific AM/PM
%S	Decimal second
%U	Decimal week of the year (starting on Sunday)
%w	Decimal Weekday (0=Sunday)
%W	Decimal week of the year (starting on Monday)
%x	Locale-specific Date
%X	Locale-specific Time
%y	2-digit year
%Y	4-digit year
%z	Offset from GMT
%Z	Time zone (character)

Now we can see that the datetime variables are converted:

```
str(times)
```

```
'data.frame':	12 obs. of  5 variables:
 $ id   : chr  "01" "02" "03" "04" ...
 $ time : POSIXct, format: "1999-03-01 11:42:00" NA "1973-08-02 03:27:41" "19
84-04-08 17:06:49" ...
 $ time2: POSIXct, format: "1962-04-24 18:25:31" "1954-03-09 15:35:26" "1962-
11-30 08:56:23" "1935-04-04 15:34:40" ...
 $ date : Date, format: "1962-12-15" "1961-09-03" "1962-04-25" "1962-11-16" .
..
 $ date2: Date, format: "2005-04-16" "1990-09-24" "1966-10-01" "1979-04-14" .
..
```

POSIX is an abbreviation for Portable Operating System Interface.

20.5. Python

Working with dates in Python is quite flexible. It is easy to convert an object to a datetime variable. We use the *pd.to_datetime* method with parameters that describe the date and time format. We also convert the variables to datetime64:

```
data="""
01,01-MAR-1999 11:42:00,24-APR-1962 18:25:31,1962-12-15,2005-04-16
02,25-DEC-2002 02:40:12,09-MAR-1954 15:35:26,1961-09-03,1990-09-24
03,02-AUG-1973 03:27:41,30-NOV-1962 08:56:23,1962-04-25,1966-10-01
04,08-APR-1984 17:06:49,04-APR-1935 15:34:40,1962-11-16,1979-04-14
05,04-FEB-2003 19:42:52,18-FEB-1963 06:53:54,1962-12-04,2013-03-17
06,02-SEP-1966 09:37:17,02-NOV-1935 16:23:38,1962-01-10,2000-07-13
07,26-SEP-1969 22:23:10,19-SEP-1964 22:15:04,1960-06-28,1986-09-18
08,10-FEB-1995 20:17:57,08-FEB-1963 06:09:53,1964-03-01,1984-05-10
09,01-SEP-1970 17:19:39,01-SEP-1970 13:58:32,1960-12-20,2018-01-05
10,20-DEC-1979 04:57:52,22-JUN-1963 05:37:12,1963-01-21,1985-04-07
11,19-JUL-2002 18:23:49,29-FEB-1961 15:41:51,1960-10-02,1960-08-21
12,04-SEP-2001 03:25:32,28-SEP-1961 03:25:32,1963-09-19,1963-09-19
"""
times =
    )
times['timea'] = pd.to_datetime(times.time, format='%d-%b-%Y %H:%M:%S',
errors='coerce').astype('datetime64[ns]')
times['time2a'] = pd.to_datetime(times.time2, format='%d-%b-%Y %H:%M:%S',
errors='coerce').astype('datetime64[ns]')
times['datea'] = pd.to_datetime(times.date, format='%Y-%m-%d',
errors='coerce').astype('datetime64[ns]')
times['date2a'] = pd.to_datetime(times.date2, format='%Y-%m-%d',
errors='coerce').astype('datetime64[ns]')
times['time_diff'] = times['timea'] - times['time2a']
times['date_diff'] = times['datea'] - times['date2a']
times
```

We now have a data frame with datetime variables. If a datetime is invalid it sets the value to NaT (not a time), see row 10:

	id	time	time2	date	date2	timea	time2a	datea	date2a	time_diff	date_diff
0	01	01-MAR-1999 11:42:00	24-APR-1962 18:25:31	1962-12-15	2005-04-16	1999-03-01 11:42:00	1962-04-24 18:25:31	1962-12-15	2005-04-16	13459 days 17:16:29	-15463 days
1	02	25-DEC-2002 02:40:12	09-MAR-1954 15:35:26	1961-09-03	1990-09-24	2002-12-25 02:40:12	1954-03-09 15:35:26	1961-09-03	1990-09-24	17822 days 11:04:46	-10613 days
2	03	02-AUG-1973 03:27:41	30-NOV-1962 08:56:23	1962-04-25	1966-10-01	1973-08-02 03:27:41	1962-11-30 08:56:23	1962-04-25	1966-10-01	3897 days 18:31:18	-1620 days
3	04	08-APR-1984 17:06:49	04-APR-1935 15:34:40	1962-11-16	1979-04-14	1984-04-08 17:06:49	1935-04-04 15:34:40	1962-11-16	1979-04-14	17902 days 01:32:09	-5993 days
4	05	04-FEB-2003 19:42:52	18-FEB-1963 06:53:54	1962-12-04	2013-03-17	2003-02-04 19:42:52	1963-02-18 06:53:54	1962-12-04	2013-03-17	14596 days 12:48:58	-18366 days
5	06	02-SEP-1966 09:37:17	02-NOV-1935 16:23:38	1962-01-10	2000-07-13	1966-09-02 09:37:17	1935-11-02 16:23:38	1962-01-10	2000-07-13	11261 days 17:13:39	-14064 days
6	07	26-SEP-1969 22:23:10	19-SEP-1964 22:15:04	1960-06-28	1986-09-18	1969-09-26 22:23:10	1964-09-19 22:15:04	1960-06-28	1986-09-18	1833 days 00:08:06	-9578 days
7	08	10-FEB-1995 20:17:57	08-FEB-1963 06:09:53	1964-03-01	1984-05-10	1995-02-10 20:17:57	1963-02-08 06:09:53	1964-03-01	1984-05-10	11690 days 14:08:04	-7375 days
8	09	01-SEP-1970 17:19:39	01-SEP-1970 13:58:32	1960-12-20	2018-01-05	1970-09-01 17:19:39	1970-09-01 13:58:32	1960-12-20	2018-01-05	0 days 03:21:07	-20835 days
9	10	20-DEC-1979 04:57:52	22-JUN-1963 05:37:12	1963-01-21	1985-04-07	1979-12-20 04:57:52	1963-06-22 05:37:12	1963-01-21	1985-04-07	6024 days 23:20:40	-8112 days
10	11	19-JUL-2002 18:23:49	29-FEB-1961 15:41:51	1960-10-02	1960-08-21	2002-07-19 18:23:49	NaT	1960-10-02	1960-08-21	NaT	42 days
11	12	04-SEP-2001 03:25:32	28-SEP-1961 03:25:32	1963-09-19	1963-09-19	2001-09-04 03:25:32	1961-09-28 03:25:32	1963-09-19	1963-09-19	14586 days 00:00:00	0 days

The datetime format is given by these parameters:

%a	Locale’s abbreviated weekday name.
%A	Locale’s full weekday name.
%b	Locale’s abbreviated month name.
%B	Locale’s full month name.
%c	Locale’s appropriate date and time representation.
%d	Day of the month as a decimal number [01,31].
%H	Hour (24-hour clock) as a decimal number [00,23].
%I	Hour (12-hour clock) as a decimal number [01,12].
%j	Day of the year as a decimal number [001,366].
%m	Month as a decimal number [01,12].
%M	Minute as a decimal number [00,59].
%p	Locale’s equivalent of either AM or PM.
%S	Second as a decimal number [00,61].
%U	Week number of the year (Sunday as the first day of the week) as a decimal number [00,53]. All days in a new year preceding the first Sunday are considered to be in week 0.
%w	Weekday as a decimal number [0(Sunday),6].
%W	Week number of the year (Monday as the first day of the week) as a decimal number [00,53]. All days in a new year preceding the first Monday are considered to be in week 0.
%x	Locale’s appropriate date representation.
%X	Locale’s appropriate time representation.
%y	Year without century as a decimal number [00,99].
%Y	Year with century as a decimal number.
%z	Time zone offset indicating a positive or negative time difference from UTC/GMT of the form +HHMM or -HHMM, where H represents decimal hour digits and M represents decimal minute digits [-23:59, +23:59].
%Z	Time zone name (no characters if no time zone exists).
%%	A literal '%' character.

21. Tabulation

We have seen earlier how we can make tables for frequencies and descriptive statistics. Quite often we need to create more advanced tables. There are several kinds of tables, and we will now look into some of them and see how we can create them in the different software packages. The results may come in different formats. Sas creates Html files as default, the other packages put the table in more or less plain text formats as default. In some of the packages it is possible to wrap layout around the table and display them in other formats. We will here only look at the default output.

There are different types of tables, and they may be rather complex. The basic tables consist of two dimensions with one or more variables in the rows and one or more variables in the columns. It should be possible to add totals and subtotals, and also to decide if they shall be placed before or after the rows or columns. When we have more than one variable in one dimension, they can be either nested or stacked. A nested variable appears as a subgroup to another variable, while stacked variables appear independently after each other.

The variables used in our tables can either be used as categorical or as measure variables. Categorical variables are used for distribution as there will be one row or column for each value of a categorical variable. Measure variables will be used for different statistical measures like sum, mean, median, etc.

We will now show some different table types and how they can be made in the different software packages. The simplest table is a two-way table with one categorical variable in the rows and one in the columns. In the cells of the table the frequency is counted:

Table 1. Simple cross tabulation with frequency counts

Var1	**Var2**		
	X	**Y**	**Z**
A	6	5	8
B	11	7	9
C	4	2	4

For each combination of the categories in the rows and columns, the number of observations/rows are counted.

Totals can be added in both dimensions. It should be possible to place them either before or after the categories. Here is an example with totals after:

Table 2. Simple cross tabulation with frequency counts and total

Var1	**Var2**			
	X	**Y**	**Z**	**Total**
A	6	5	8	19
B	11	7	9	27
C	4	2	4	10
Total	21	14	21	56

We can have nested variables in the rows and/or columns:

Table 3. Cross tabulation with nested variables in the rows, frequency counts and total

Var1	Var3	Var2			
		X	Y	Z	Total
A	G	4	1	5	10
	H	2	4	3	9
B	G	8	3	5	16
	H	3	4	4	11
C	G	4	0	1	5
	H	0	2	3	5
Total		21	14	21	56

Here we have nested variables in both dimensions:

Table 4. Cross tabulation with nested variables in the rows and columns, frequency counts and total

Var1	Var3	Var2						Total
		X		Y		Z		
		Var4		Var4		Var4		
		M	P	M	P	M	P	
A	G	2	2	1	0	2	3	10
	H	2	0	0	4	2	1	9
B	G	2	6	2	1	2	3	16
	H	1	2	3	1	2	2	11
C	G	3	1	0	0	1	0	5
	H	0	0	1	1	1	2	5
Total		10	11	7	7	10	11	56

Now we add subtotals. We see that the total is also calculated for *Var3* in the rows and *Var4* in the columns. The subtotals of these will add up to the grand total:

Table 5. Cross tabulation with two nested variables in the rows and two nested in the columns, frequency counts, total and subtotals

Var1	**Var3**	**Var2**											
		X			**Y**			**Z**			**Total**		
		Var4			**Var4**			**Var4**			**Var4**		
		M	**P**	**Subtotal**	**M**	**P**	**Subtotal**	**M**	**P**	**Subtotal**	**M**	**P**	**Subtotal**
A	**G**	2	2	4	1	0	1	2	3	5	5	5	10
	H	2	0	2	0	4	4	2	1	3	4	5	9
	Subtotal	4	2	6	1	4	5	4	4	8	9	10	19
B	**G**	2	6	8	2	1	3	2	3	5	6	10	16
	H	1	2	3	3	1	4	2	2	4	6	5	11
	Subtotal	3	8	11	5	2	7	4	5	9	12	15	27
C	**G**	3	1	4	0	0	0	1	0	1	4	1	5
	H	0	0	0	1	1	2	1	2	3	2	3	5
	Subtotal	3	1	4	1	1	2	2	2	4	6	4	10
Total	**G**	7	9	16	3	1	4	5	6	11	15	16	31
	H	3	2	5	4	6	10	5	5	10	12	13	25
	Subtotal	10	11	21	7	7	14	10	11	21	27	29	56

Instead of nested variables, we can have them stacked:

Table 6. Cross tabulation with stacked variables in rows and columns, frequency counts and total

Var1	**Var2**			**Var4**		
	X	**Y**	**Z**	**M**	**P**	**Total**
A	6	5	8	9	10	19
B	11	7	9	12	15	27
C	4	2	4	6	4	10
Var3						
G	16	4	11	15	16	31
H	5	10	10	12	13	25
Total	21	14	21	27	29	56

Instead of counting the frequency, we can count the sum, mean etc. of a measure variable. We can still have categorical variables in the rows and columns. Her we have chosen to put the sum of the *Var6* variable into the cells of the tables:

Table 7. Cross tabulation with sum of measure variable and totals

Var1	Var2			
	X	Y	Z	Total
	Var6	Var6	Var6	
	Sum	Sum	Sum	Sum
A	21	33	11	65
B	24	26	15	65
C	27	21	12	60
Total	72	80	38	190

We should be able to calculate percentages in our tables, both total, row and columns percentages:

Table 8. Cross tabulation with percent of total frequency counts

Var1	Var2			Total
	X	Y	Z	
A	10,7	8,9	14,3	33,9
B	19,6	12,5	16,1	48,2
C	7,1	3,6	7,1	17,9
Total	37,5	25,0	37,5	100,0

Table 9. Cross tabulation with percent of row frequency counts

Var1	Var2			Total
	X	Y	Z	
A	31,6	26,3	42,1	100,0
B	40,7	25,9	33,3	100,0
C	40,0	20,0	40,0	100,0
Total	37,5	25,0	37,5	100,0

Table 10. Cross tabulation with percent of column frequency counts

Var1	Var2			Total
	X	Y	Z	
A	28,6	35,7	38,1	33,9
B	52,4	50,0	42,9	48,2
C	19,0	14,3	19,0	17,9
Total	100,0	100,0	100,0	100,0

Finally, different counts can be shown in the same table. Here we have both frequencies and column frequencies:

Table 11. Cross tabulation with percent of column frequency counts

Var1	N				%			
	Var2				Var2			Total
	X	Y	Z	Total	X	Y	Z	
A	6	5	8	19	28,6	35,7	38,1	33,9
B	11	7	9	27	52,4	50,0	42,9	48,2
C	4	2	4	10	19,0	14,3	19,0	17,9
Total	21	14	21	56	100,0	100,0	100,0	100,0

These are just some examples on what kind of tables it should be possible to make in a tabulation procedure. Now we will look at how these tables can be made in the different software packages.

21.1. Sas

In Sas, there are two procedures that is suitable for making tables: *Proc tabulate* and *Proc report*. The default output is sent to the result window as an html file. This means the tables looks fine and they can easily be exported to for instance Excel. Here, we will only look at *Proc tabulate*.

We use the *Proc tabulate* statement to choose the dataset to create the table from. We have to define how the variables we use in the table is to be used and we do that with the *Class* and *Var* statements. Variables mentioned in the *Class* statement will be used as categorical variables. Variables mentioned in the *Var* statement will be used as measure variables. The table layout will be defined in the *Table* statement.

Here is the syntax to create a two-way table with frequency counts:

```
proc tabulate data=mdg.mdgperson_nodup f=15. missing;
 class state urbrur;
 table state
        ,
          urbrur
          /misstext='-';
 title "Table 1. Persons by state and location";
run;
```

As we define both *state* and *urbrur* in the *Class* statement, they will be categorical variables. In the *Table* statement the comma divides the dimensions from each other. What is placed before the comma will end up in the rows and after the comma in the columns. As we have not explicitly defined what to count, Sas will count the frequencies and show that with the column header *N*. We can add some options for the table after the slash. Here we have added the option *Misstext* to tell that we want to fill empty cells with a dash. The table will be like this:

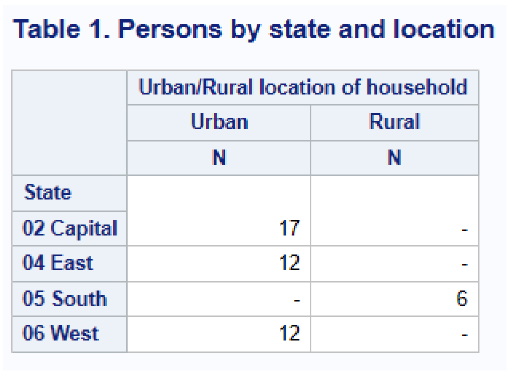


Table 1. Persons by state and location

	Urban/Rural location of household	
	Urban	Rural
	N	N
State		
02 Capital	17	-
04 East	12	-
05 South	-	6
06 West	12	-

We can add total to the table with the *All* option. We also tell that we want to count the frequencies with the *N* option, and then we can also tell that we don't want any column header for it. The syntax is now like this:

```
proc tabulate data=mdg.mdgperson_nodup f=15. missing;
 class state urbrur;
 table all='Total' state=''
        ,
          (all='Total' urbrur='')*n=''
          /misstext='-';
 title "Table 2. Persons by state and location. With totals";
run;
```

We have put the totals before the categories:

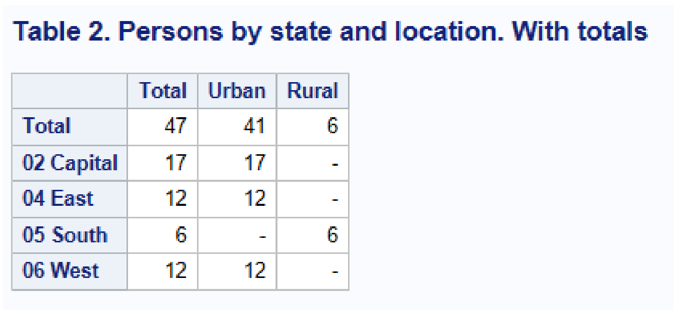


Table 2. Persons by state and location. With totals

	Total	Urban	Rural
Total	47	41	6
02 Capital	17	17	-
04 East	12	12	-
05 South	6	-	6
06 West	12	12	-

Now we will have two nested variables in the rows. We add the gender variable (*b4*) to the class statement and include the variable in the rows in the table statement. To nest the variables, we insert an * sign between the two variables:

```
proc tabulate data=mdg.mdgperson_nodup f=15. missing;
 class state urbrur b4;
 table all='Total' state=''*b4=''
       ,
         (all='Total' urbrur='')*n=''
         /misstext='-' nocellmerge;
 title "Table 3. Persons by state, sex and location. Nested";
run;
```

The table have now two nested variables in the rows. For each value of the *state* variable, which is mentioned first, we will have a gender distribution. The sub-group *b4* is not distributed for the total, it is only a grand total:

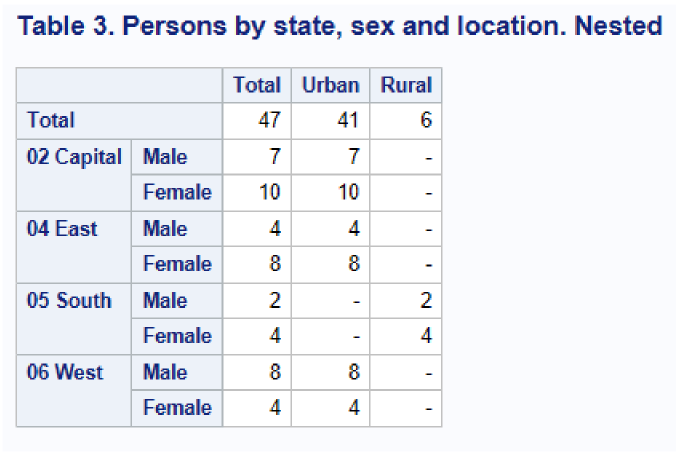


Table 3. Persons by state, sex and location. Nested

		Total	Urban	Rural
Total		47	41	6
02 Capital	Male	7	7	-
	Female	10	10	-
04 East	Male	4	4	-
	Female	8	8	-
05 South	Male	2	-	2
	Female	4	-	4
06 West	Male	8	8	-
	Female	4	4	-

We will now add a nested variable, *b6*, in the columns as well. It is done the same way as previously, except the new variable is inserted in the columns of the table definition:

```
proc tabulate data=mdg.mdgperson_nodup f=15. missing;
 class state urbrur b4 b6;
 table all='Total' state=''*b4=''
       ,
         (all='Total' b6=''*urbrur='')*n=''
         /misstext='-' nocellmerge;
 title "Table 4. Persons by state, sex, civil status and location. Nested";
run;
```

The table show the nesting in both dimensions. Categories with no values are excluded, we can see that because there is no Rural column for Missing values or Married - Polygamy:

Table 4. Persons by state, sex, civil status and location. Nested

		Total	Missing	Never married		Married - monogamy		Married - polygamy
			Urban	Urban	Rural	Urban	Rural	Urban
Total		47	12	15	3	10	3	4
02 Capital	Male	7	2	1	-	2	-	2
	Female	10	4	3	-	3	-	-
04 East	Male	4	1	2	-	1	-	-
	Female	8	3	2	-	1	-	2
05 South	Male	2	-	-	1	-	1	-
	Female	4	-	-	2	-	2	-
06 West	Male	8	2	5	-	1	-	-
	Female	4	-	2	-	2	-	-

We can now add subtotals to our nested table. We add the *all* option for the subtotals and introduce parenthesis for adding have the distribution of the nested variables for the totals as well:

```
proc tabulate data=mdg.mdgperson_nodup f=15. missing;
 class state urbrur b4 b6;
 table (all='Total' state='')*(all='Both genders' b4='')
        ,
          (all='Total' b6='')*(all='Both locations' urbrur='')*n=''
          /misstext='-' nocellmerge;
 title "Table 5. Persons by state, sex, civil status and location. Nested
with totals";
run;
```

Our table is now extended:

Table 5. Persons by state, sex, civil status and location. Nested with totals

		Total			Missing		Never married			Married - monogamy			Married - polygamy	
		Both locations	Urban	Rural	Both locations	Urban	Both locations	Urban	Rural	Both locations	Urban	Rural	Both locations	Urban
Total	Both genders	47	41	6	12	12	18	15	3	13	10	3	4	4
	Male	21	19	2	5	5	9	8	1	5	4	1	2	2
	Female	26	22	4	7	7	9	7	2	8	6	2	2	2
02 Capital	Both genders	17	17	-	6	6	4	4	-	5	5	-	2	2
	Male	7	7	-	2	2	1	1	-	2	2	-	2	2
	Female	10	10	-	4	4	3	3	-	3	3	-	-	-
04 East	Both genders	12	12	-	4	4	4	4	-	2	2	-	2	2
	Male	4	4	-	1	1	2	2	-	1	1	-	-	-
	Female	8	8	-	3	3	2	2	-	1	1	-	2	2
05 South	Both genders	6	-	6	-	-	3	-	3	3	-	3	-	-
	Male	2	-	2	-	-	1	-	1	1	-	1	-	-
	Female	4	-	4	-	-	2	-	2	2	-	2	-	-
06 West	Both genders	12	12	-	2	2	7	7	-	3	3	-	-	-
	Male	8	8	-	2	2	5	5	-	1	1	-	-	-
	Female	4	4	-	-	-	2	2	-	2	2	-	-	-

Now we want to have our variables stacked after each other instead of nested within each other. To stack the variables, we simply drop the * sign between the listed variables in the table statement:

```
proc tabulate data=mdg.mdgperson_nodup f=15. missing;
 class state urbrur b4 b6;
 table all='Total' state b4
        ,
          (all='Total' b6 urbrur)*n=''
          /misstext='-' nocellmerge;
 title "Table 6. Persons by state, sex, civil status and location. Stacked";
run;
```

We see that our variables are now stacked, both in the rows and columns:

Table 6. Persons by state, sex, civil status and location. Stacked

	Total	Civil status				Urban/Rural location of household	
		Missing	Never married	Married - monogamy	Married - polygamy	Urban	Rural
Total	47	12	18	13	4	41	6
State							
02 Capital	17	6	4	5	2	17	-
04 East	12	4	4	2	2	12	-
05 South	6	-	3	3	-	-	6
06 West	12	2	7	3	-	12	-
Sex							
Male	21	5	9	5	2	19	2
Female	26	7	9	8	2	22	4

Now we will introduce a measure variable. It shall be defined in the *Var* statement. When we use it, we should also connect it to a measurement, like *sum* or *mean*. If not, the sums will be calculated. We want to calculate the average age (*b5*) of the persons distributed by *state* and *b4*:

```
proc tabulate data=mdg.mdgperson_nodup f=15. missing;
 class state b4 ;
 var b5;
 table all='Total' state=''
          ,
           (all='Total' b4='')*b5=''*mean=''
           /misstext='-' nocellmerge;
 title "Table 7. Average age of persons by state and sex";
run;
```

The connection is done with the * sign. The table cells will now contain average age:

Table 7. Average age of persons by state and sex

	Total	Male	Female
Total	21	22	21
02 Capital	19	24	16
04 East	18	18	17
05 South	34	40	32
06 West	22	18	28

Now we will look at three ways to calculate percentages, of grand total, rows and columns. We choose between these with either *pctn*, *rowpctn* or *colpctn*:

```
proc tabulate data=mdg.mdgperson_nodup f=15. missing;
 class state b4 ;
 table all='Total' state=''
          ,
           (all='Total' b4='')*pctn=''
           /misstext='-' nocellmerge;
 title "Table 8. Persons by state and sex. Percent of total";
run;

proc tabulate data=mdg.mdgperson_nodup f=15. missing;
 class state b4 ;
 table all='Total' state=''
```

```
       ,
         (all='Total' b4='')*rowpctn=''
         /misstext='-' nocellmerge;
 title "Table 9. Persons by state and sex. Percent of rows";
run;

proc tabulate data=mdg.mdgperson_nodup f=15. missing;
 class state b4 ;
 table all='Total' state=''
       ,
         (all='Total' b4='')*colpctn=''
         /misstext='-' nocellmerge;
 title "Table 10. Persons by state and sex. Percent of columns";
run;
```

The tables will be like this:

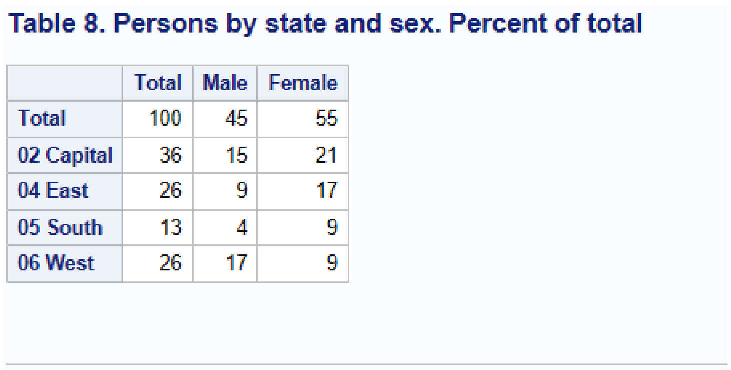


Table 8. Persons by state and sex. Percent of total

	Total	Male	Female
Total	100	45	55
02 Capital	36	15	21
04 East	26	9	17
05 South	13	4	9
06 West	26	17	9

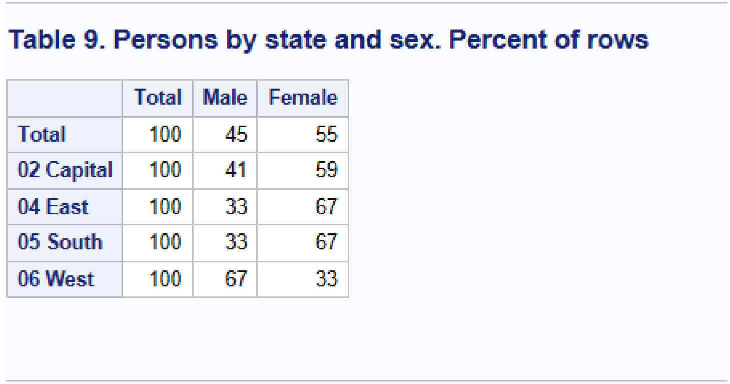


Table 9. Persons by state and sex. Percent of rows

	Total	Male	Female
Total	100	45	55
02 Capital	100	41	59
04 East	100	33	67
05 South	100	33	67
06 West	100	67	33

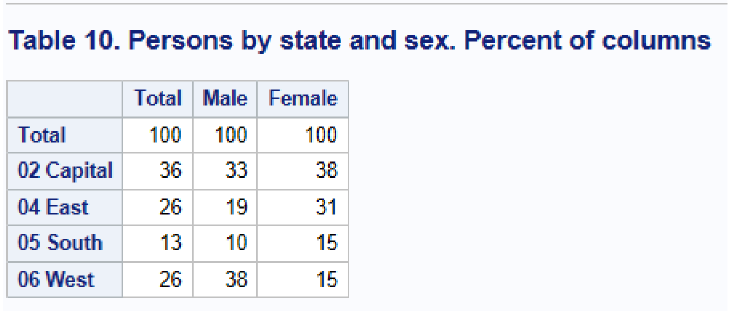


Table 10. Persons by state and sex. Percent of columns

	Total	Male	Female
Total	100	100	100
02 Capital	36	33	38
04 East	26	19	31
05 South	13	10	15
06 West	26	38	15

Finally, we can combine absolute figures with percentages. Her we choose all the absolute figures in the first columns and then the column percentages:

```
proc tabulate data=mdg.mdgperson_nodup f=15. missing;
 class state b4 ;
 table all='Total' state=''
       ,
         (n rowpctn='%')*(all='Total' b4='')
         /misstext='-' nocellmerge;
 title "Table 11. Persons by state and sex. Absolute figures and percent of
rows";
run;
```

This is the table produced:

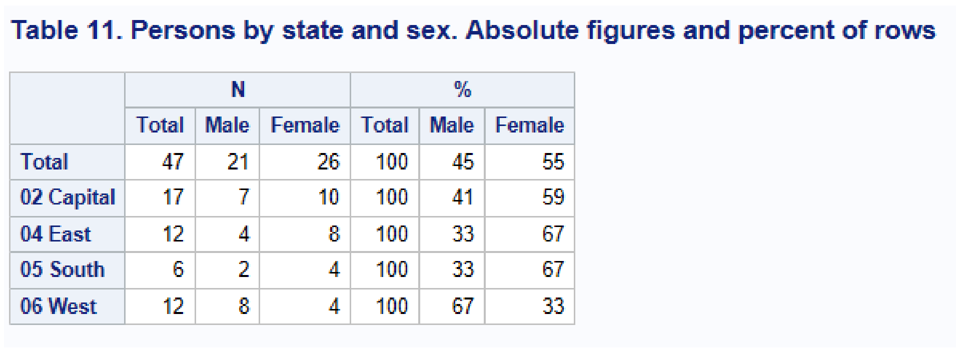


Table 11. Persons by state and sex. Absolute figures and percent of rows

	N			%		
	Total	Male	Female	Total	Male	Female
Total	47	21	26	100	45	55
02 Capital	17	7	10	100	41	59
04 East	12	4	8	100	33	67
05 South	6	2	4	100	33	67
06 West	12	8	4	100	67	33

It is possible to create other kinds of tables with *Proc tabulate*, so it is a useful procedure to learn.

21.2. Spss

In Spss the *CTABLES* command is suitable for tabulation. It is found under the Analyze➔Tables➔Custom Tables menu. Here we can generate syntax which can be executed, or we can write our own syntax. The default output is sent formatted to the Output Window. From there it is easy to export the tables into for instance Excel. We just right click on the table and choose *Copy* and then *Excel spreadsheet*.

In *CTABLES* we can use these subcommands for our basic tabulation:

- VLABELS Decide how to display value labels
- TABLE Define the table layout. The rows are defined first. Then, after BY, the columns are defined
- CATEGORIES Describe options for the categorical variables
- TITLES The title of the table

The variables can be defined as Categorical (C), Ordinal (O) or Scale (S). This can be done within the Table subcommand. Only Scale variables may be used for measures.

In our first example we have decided to show the labels of the variables *state* and *urbrur*. In the table we put *state* in the rows and *urbrur* in the columns. In the cells of the table, we put the count which will count the number of cases with the different values for these categorical variables. In the *Categories* subcommand we say that both variables are to be sorted in ascending order by the value of the variables. We also will exclude user missing values if there were any present in the variables used (system missing values are always excluded):

```
GET FILE='mdgperson_nodup.sav'.
CTABLES
  /VLABELS VARIABLES=state urbrur DISPLAY=LABEL
  /TABLE state [COUNT F40.0] BY urbrur
  /CATEGORIES VARIABLES=state urbrur ORDER=A KEY=VALUE EMPTY=EXCLUDE
  /TITLES
    TITLE='Table 1. Persons by state and location'.
```

When we run, the table is sent to the Output window and looks like this:

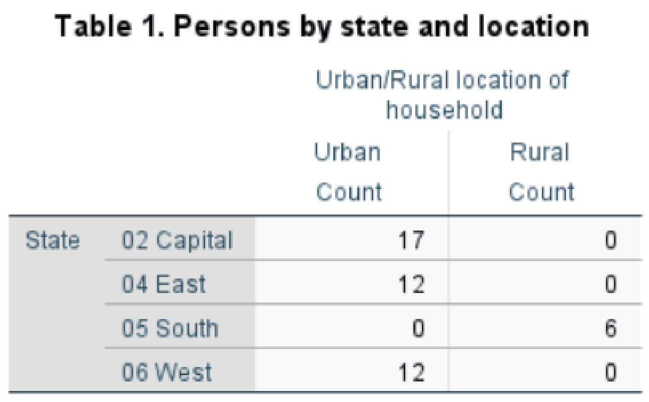


Table 1. Persons by state and location

		Urban/Rural location of household	
		Urban	Rural
		Count	Count
State	02 Capital	17	0
	04 East	12	0
	05 South	0	6
	06 West	12	0

We can now add the totals to our table and choose to have them before the variables. For this, we use the *Total=yes* and *Position=before* options:

```
CTABLES
  /VLABELS VARIABLES=state urbrur DISPLAY=NONE
  /TABLE state [COUNT F40.0] BY urbrur
  /SLABELS VISIBLE=NO
  /CATEGORIES VARIABLES=state urbrur ORDER=A KEY=VALUE EMPTY=EXCLUDE
TOTAL=YES POSITION=BEFORE
  /TITLES
    TITLE='Table 2. Persons by state and location. With totals'.
```

Totals are now added to our table:

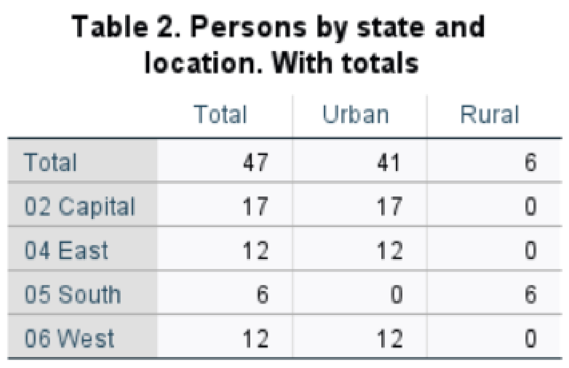


Table 2. Persons by state and location. With totals

	Total	Urban	Rural
Total	47	41	6
02 Capital	17	17	0
04 East	12	12	0
05 South	6	0	6
06 West	12	12	0

We can add a nested variable, *b4*, to our rows. Then we use the > sign to do the nesting. We see that we can have more than one *Categories* subcommand. When parameters differ between the categorical variables, we need more *Categories* subcommands:

```
CTABLES
  /VLABELS VARIABLES=state b4 urbrur DISPLAY=NONE
  /TABLE state > b4 [COUNT F40.0] BY urbrur
  /SLABELS VISIBLE=NO
  /CATEGORIES VARIABLES=state urbrur ORDER=A KEY=VALUE EMPTY=EXCLUDE
TOTAL=YES POSITION=BEFORE
  /CATEGORIES VARIABLES=b4 ORDER=A KEY=VALUE EMPTY=EXCLUDE
  /TITLES
    TITLE='Table 3. Persons by state, sex and location. Nested'.
```

The table shows the nested variables in the rows. We see that the subgroup *b4* is also used for the total. However, there is no grand total in this table:

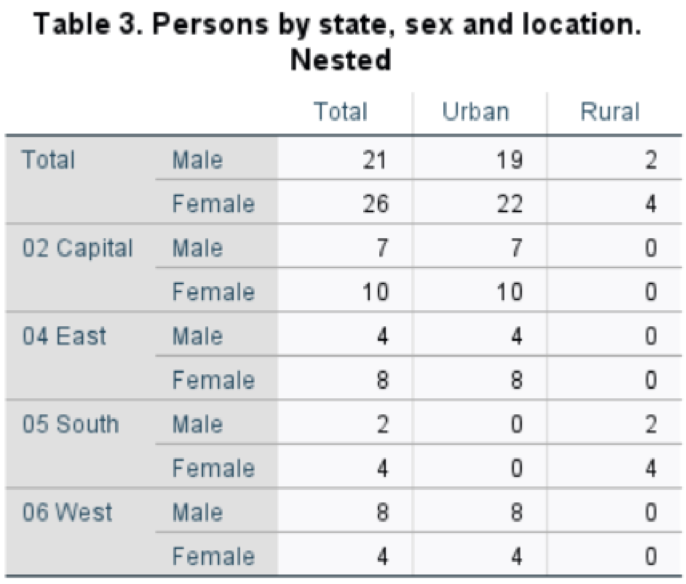


Table 3. Persons by state, sex and location. Nested

		Total	Urban	Rural
Total	Male	21	19	2
	Female	26	22	4
02 Capital	Male	7	7	0
	Female	10	10	0
04 East	Male	4	4	0
	Female	8	8	0
05 South	Male	2	0	2
	Female	4	0	4
06 West	Male	8	8	0
	Female	4	4	0

Now we want to add the variable *b6* nested under to the *urbrur* variable in the columns. *b6* has missing values and to include them in the table they must be set to user missing or a valid value. We set them to user missing first and then make the table:

```
IF (missing(b6) = 1) b6 = 9.
MISSING VALUES b6 (9).

CTABLES
  /VLABELS VARIABLES=state b4 b6 urbrur DISPLAY=NONE
  /TABLE state > b4 [COUNT F40.0] BY b6 > urbrur
  /SLABELS VISIBLE=NO
  /CATEGORIES VARIABLES=state b6 ORDER=A EMPTY=EXCLUDE TOTAL=YES
POSITION=BEFORE MISSING=include
  /CATEGORIES VARIABLES=b4 ORDER=A KEY=VALUE EMPTY=EXCLUDE TOTAL=NO
LABEL='Both sexes'
    POSITION=BEFORE
  /CATEGORIES VARIABLES=urbrur ORDER=A KEY=VALUE EMPTY=EXCLUDE TOTAL=NO
LABEL='Both locations'
    POSITION=BEFORE
  /TITLES
    TITLE='Table 4. Persons by state, sex, civil status and location.
Nested'.
```

For the variable b6, we have added the *missing=include* option in the *Categories* subcommand. The user missing values will now be included in the table:

Table 4. Persons by state, sex, civil status and location. Nested

		Total		Never married		Married – monogamy		Married – polygamy	N/A
		Urban	Rural	Urban	Rural	Urban	Rural	Urban	Urban
Total	Male	19	2	8	1	4	1	2	5
	Female	22	4	7	2	6	2	2	7
02 Capital	Male	7	0	1	0	2	0	2	2
	Female	10	0	3	0	3	0	0	4
04 East	Male	4	0	2	0	1	0	0	1
	Female	8	0	2	0	1	0	2	3
05 South	Male	0	2	0	1	0	1	0	0
	Female	0	4	0	2	0	2	0	0
06 West	Male	8	0	5	0	1	0	0	2
	Female	4	0	2	0	2	0	0	0

In the next table we want to add sub-totals. We do that by changing the *Total* parameter to *Yes* in the *Categories* commands. We have also made different labels for the sub-totals:

```
CTABLES
  /VLABELS VARIABLES=state b4 b6 urbrur DISPLAY=NONE
  /TABLE state > b4 [COUNT F40.0] BY b6 > urbrur
  /SLABELS VISIBLE=NO
  /CATEGORIES VARIABLES=state b6 ORDER=A EMPTY=EXCLUDE TOTAL=YES
POSITION=BEFORE MISSING=include
  /CATEGORIES VARIABLES=b4 ORDER=A KEY=VALUE EMPTY=EXCLUDE TOTAL=YES
LABEL='Both sexes'
    POSITION=BEFORE
  /CATEGORIES VARIABLES=urbrur ORDER=A KEY=VALUE EMPTY=EXCLUDE TOTAL=YES
LABEL='Both locations'
    POSITION=BEFORE
  /TITLES
    TITLE='Table 5. Persons by state, sex, civil status and location. Nested
with totals'.
```

The sub-totals are now added. We see that for the total we have both the grand total and the totals within the *b4* and *urbrur* variables:

Table 5. Persons by state, sex, civil status and location. Nested with totals

		Total			Never married			Married – monogamy			Married – polygamy		N/A	
		Both locations	Urban	Rural	Both locations	Urban	Rural	Both locations	Urban	Rural	Both locations	Urban	Both locations	Urban
Total	Both sexes	47	41	6	18	15	3	13	10	3	4	4	12	12
	Male	21	19	2	9	8	1	5	4	1	2	2	5	5
	Female	26	22	4	9	7	2	8	6	2	2	2	7	7
02 Capital	Both sexes	17	17	0	4	4	0	5	5	0	2	2	6	6
	Male	7	7	0	1	1	0	2	2	0	2	2	2	2
	Female	10	10	0	3	3	0	3	3	0	0	0	4	4
04 East	Both sexes	12	12	0	4	4	0	2	2	0	2	2	4	4
	Male	4	4	0	2	2	0	1	1	0	0	0	1	1
	Female	8	8	0	2	2	0	1	1	0	2	2	3	3
05 South	Both sexes	6	0	6	3	0	3	3	0	3	0	0	0	0
	Male	2	0	2	1	0	1	1	0	1	0	0	0	0
	Female	4	0	4	2	0	2	2	0	2	0	0	0	0
06 West	Both sexes	12	12	0	7	7	0	3	3	0	0	0	2	2
	Male	8	8	0	5	5	0	1	1	0	0	0	2	2
	Female	4	4	0	2	2	0	2	2	0	0	0	0	0

We would now like to stack the variables in the row and columns. Then we use the + sign instead of the > sign in the Tables subcommand:

```
CTABLES
  /VLABELS VARIABLES=state b4 b6 urbrur DISPLAY=LABEL
  /TABLE state [COUNT F40.0] + b4 [COUNT F40.0] BY b6 + urbrur
  /SLABELS VISIBLE=NO
  /CATEGORIES VARIABLES=state ORDER=A KEY=VALUE EMPTY=EXCLUDE TOTAL=YES
POSITION=BEFORE
  /CATEGORIES VARIABLES=b4 urbrur ORDER=A KEY=VALUE EMPTY=EXCLUDE
  /CATEGORIES VARIABLES=b6 ORDER=A KEY=VALUE EMPTY=EXCLUDE TOTAL=YES
POSITION=BEFORE MISSING=INCLUDE
  /TITLES
    TITLE='Table 6. Persons by state, sex, civil status and location.
Stacked'.
```

Now the variables follow each other, they are stacked:

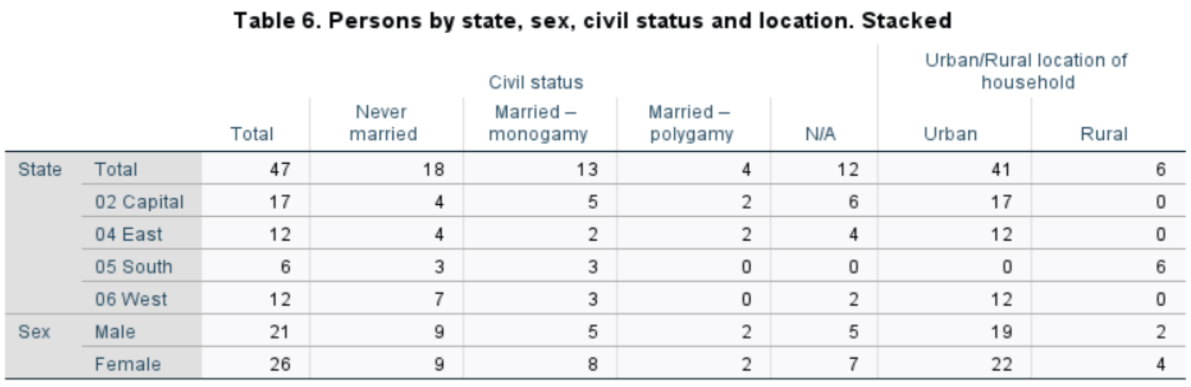


Table 6. Persons by state, sex, civil status and location. Stacked

			Civil status				Urban/Rural location of household	
		Total	Never married	Married – monogamy	Married – polygamy	N/A	Urban	Rural
State	Total	47	18	13	4	12	41	6
	02 Capital	17	4	5	2	6	17	0
	04 East	12	4	2	2	4	12	0
	05 South	6	3	3	0	0	0	6
	06 West	12	7	3	0	2	12	0
Sex	Male	21	9	5	2	5	19	2
	Female	26	9	8	2	7	22	4

We will now introduce *b5* as a scale variable, so we can have average age in our table cells. Now we nest b4 with *b5* and choose the mean of the *b5* variable. We include [S] after the variable name to make sure it is used as a scale variable. For *b4* we have added [C] to be sure it is used as a categorical variable. This is only necessary if the variables are not defined the way we want to use them:

```
CTABLES
  /VLABELS VARIABLES=state b4 b5 DISPLAY=NONE
  /TABLE state BY b4 [C] > b5 [S] [MEAN F40.0]
  /SLABELS VISIBLE=NO
  /CATEGORIES VARIABLES=state b4 ORDER=A KEY=VALUE EMPTY=EXCLUDE TOTAL=YES
POSITION=BEFORE
  /TITLES
    TITLE='Table 7. Average age of persons by state and sex'.
```

The table with the average ages is here:

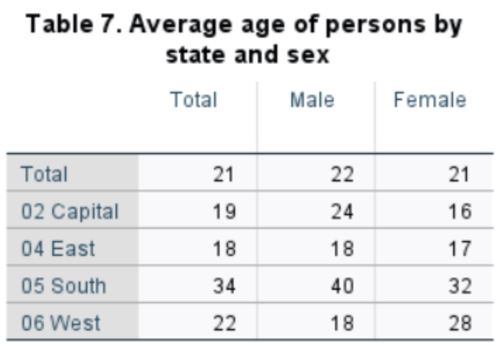


Table 7. Average age of persons by state and sex

	Total	Male	Female
Total	21	22	21
02 Capital	19	24	16
04 East	18	18	17
05 South	34	40	32
06 West	22	18	28

To make percentage tables we use the *tablepct.count*, *rowpct.count* and the *colpct.count* measures. This will give us tables with percentages made by the grand total, the row totals and the column totals:

```
CTABLES
  /VLABELS VARIABLES=state b4 DISPLAY=NONE
  /TABLE state BY b4 [TABLEPCT.COUNT F40.0]
  /SLABELS VISIBLE=NO
  /CATEGORIES VARIABLES=state b4 ORDER=A KEY=VALUE EMPTY=EXCLUDE TOTAL=YES
POSITION=BEFORE
  /TITLES
    TITLE='Table 8. Persons by state and sex. Percent of total'.
CTABLES
  /VLABELS VARIABLES=state b4 DISPLAY=NONE
  /TABLE state BY b4 [ROWPCT.COUNT F40.0]
  /SLABELS VISIBLE=NO
  /CATEGORIES VARIABLES=state b4 ORDER=A KEY=VALUE EMPTY=EXCLUDE TOTAL=YES
POSITION=BEFORE
  /TITLES
    TITLE='Table 9. Persons by state and sex. Percent of rows'.

CTABLES
  /VLABELS VARIABLES=state b4 DISPLAY=NONE
  /TABLE state BY b4 [COLPCT.COUNT F40.0]
  /SLABELS VISIBLE=NO
  /CATEGORIES VARIABLES=state b4 ORDER=A KEY=VALUE EMPTY=EXCLUDE TOTAL=YES
POSITION=BEFORE
  /TITLES
    TITLE='Table 10. Persons by state and sex. Percent of columns'.
```

The percentage tables:

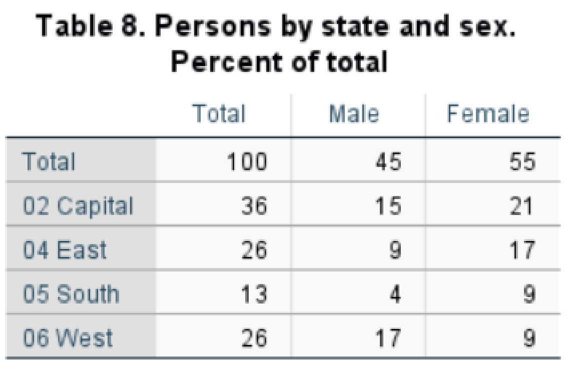


Table 8. Persons by state and sex. Percent of total

	Total	Male	Female
Total	100	45	55
02 Capital	36	15	21
04 East	26	9	17
05 South	13	4	9
06 West	26	17	9

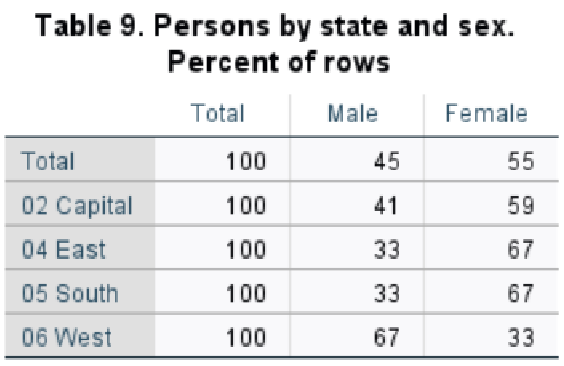


Table 9. Persons by state and sex. Percent of rows

	Total	Male	Female
Total	100	45	55
02 Capital	100	41	59
04 East	100	33	67
05 South	100	33	67
06 West	100	67	33

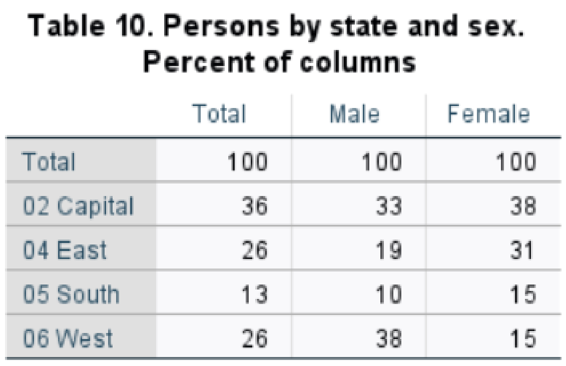


Table 10. Persons by state and sex. Percent of columns

	Total	Male	Female
Total	100	100	100
02 Capital	36	33	38
04 East	26	19	31
05 South	13	10	15
06 West	26	38	15

Finally, we will combine absolute figures with column percentages:

```
CTABLES
  /VLABELS VARIABLES=state b4 DISPLAY=NONE
  /TABLE state BY b4 [COUNT 'N' F40.0] + b4 [COLPCT.COUNT '%' F40.0]
  /CATEGORIES VARIABLES=state ORDER=A KEY=VALUE EMPTY=EXCLUDE TOTAL=YES
POSITION=BEFORE
  /CATEGORIES VARIABLES=b4 ORDER=A KEY=VALUE EMPTY=INCLUDE TOTAL=YES
POSITION=BEFORE
  /TITLES
   TITLE='Table 11. Persons by state and sex. Absolute figures and percent of
columns'.
```

Unfortunately, the measure texts (*N, %*) are placed under the variable and repeated. It would be better to have them once above the variable texts. It does not seem to be possible to change this in an easy way (one way is to copy the *b4* variable to a new name, give them different labels and use one of them for the absolute figures and the other one for the percentages):

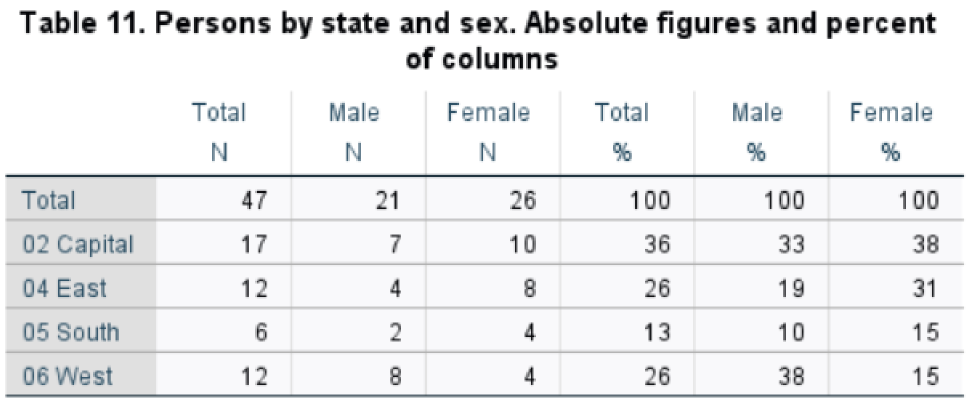


Table 11. Persons by state and sex. Absolute figures and percent of columns

	Total	Male	Female	Total	Male	Female
	N	N	N	%	%	%
Total	47	21	26	100	100	100
02 Capital	17	7	10	36	33	38
04 East	12	4	8	26	19	31
05 South	6	2	4	13	10	15
06 West	12	8	4	26	38	15

There are several more ways to use the *CTABLES* command. Hence, it is very useful for presenting results in Spss.

21.3. Stata

In Stata we can use the procedures *table, tabulate* and *tab2* for cross tabulation. These procedures are not especially flexible but can be useful to create basic tables. The table is sent to the *Log and listing window*. To copy the table to another format we can mark the table, right click, and choose *Copy Table* or *Copy table as Html*. Then we can paste it into for instance Excel.

The *table*, *tabulate (tab)* and *tab2* procedures syntax are short and concise. It is not possible to add titles to table within these procedures. Stata operates with rows, superrows, columns and supercolumns. Supercolumns are used for nesting variables in the columns and the by option to nest variables in the rows (superrows). It is not possible stack variables in either dimension.

We will start with a simple table with one categorical variable in each dimension. We just mention the two variables in the table procedure and the first (*state*) will be placed in the rows and the second (*urbrur*) in the columns. We add the missing option to display a dot in empty cells:

```
table state urbrur, missing
```

This is the table:

```
           | Urban/Rural
           | location of
           |  household
     State | Urban   Rural
-----------+--------------
02 Capital |    17       .
   04 East |    12       .
  05 South |     .       6
   06 West |    12       .
-----------+--------------
```

Next, we will add totals. They will always appear at the end. We just change to *tab* procedure and the totals will be included in our table:

`tab state urbrur`

Now we have totals in both dimensions. Empty cells are filled with zeros:

```
           | Urban/Rural location
           |     of household
     State |     Urban      Rural |     Total
-----------+----------------------+----------
02 Capital |        17          0 |        17
   04 East |        12          0 |        12
  05 South |         0          6 |         6
   06 West |        12          0 |        12
-----------+----------------------+----------
     Total |        41          6 |        47
```

Now we want to have two nested variables in the rows. We use the *table* procedure and add the row and col options to have totals added. We use the variable *state* in the by option to have it as the first variable (superrows) in the rows. To have dots instead of nothing in empty cells we use the missing option:

`table b4 urbrur, by(state) row col missing`

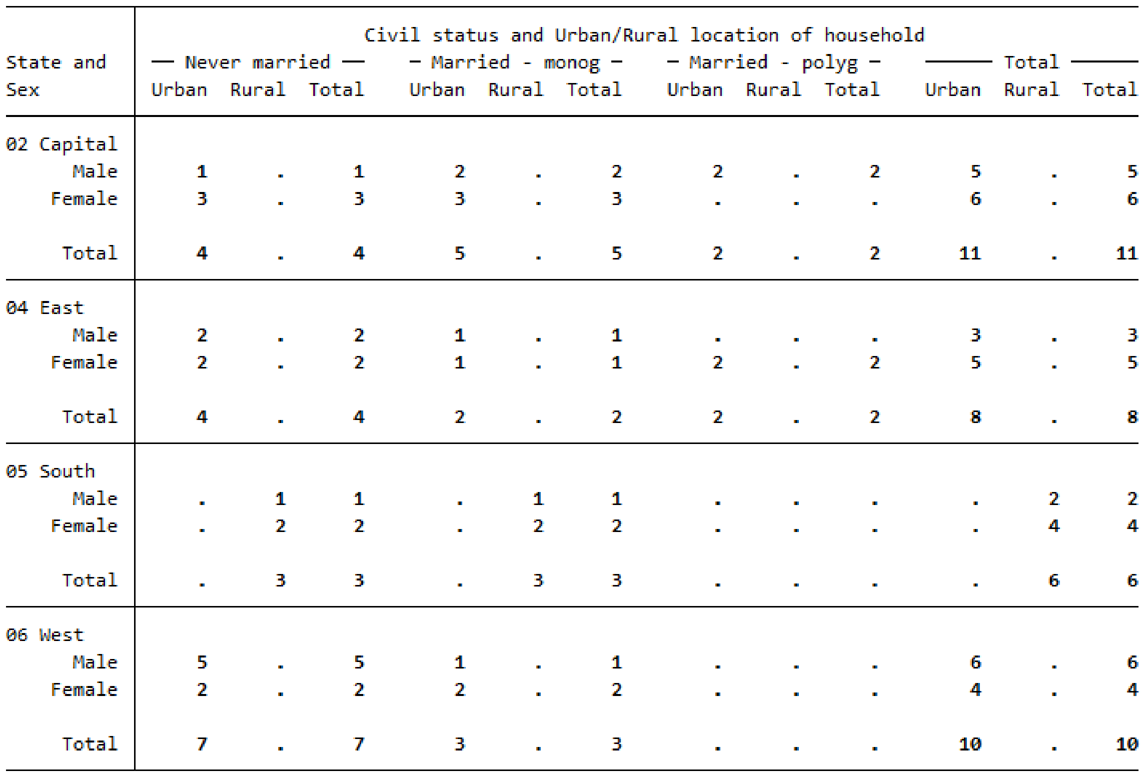
We have now a table with two nested variables in the rows. The totals added are actually subtotals for each of the states. The grand total is not included in the table:

```
------------------------------------
           | Urban/Rural location
State and  |    of household
Sex        |  Urban   Rural   Total
-----------+------------------------
02 Capital |
      Male |      7       .       7
    Female |     10       .      10
           |
     Total |     17       .      17
-----------+------------------------
04 East    |
      Male |      4       .       4
    Female |      8       .       8
           |
     Total |     12       .      12
-----------+------------------------
05 South   |
      Male |      .       2       2
    Female |      .       4       4
           |
     Total |      .       6       6
-----------+------------------------
06 West    |
      Male |      8       .       8
    Female |      4       .       4
           |
     Total |     12       .      12
------------------------------------
```

We now want nested variables in the columns as well. In the table command, we start we the nested variable *b4*. Then we name the two variables in the columns. *Urbrur* is named first and will be the nested one. *B6* is then mentioned and will be the supercolumn. After the comma, we use the by option to name the *state* variable in the rows, as the superrow variable. We also add subtotals with the row (row totals), col (column totals) and sc (supercolumn totals) options. Finally, we include the missing option to place dots in empty cells:

```
table b4 urbrur b6, by(state) row col sc missing
```

The table is a little bit strange because there are columns for the total of the nested variable, *urbrur*, independent of the *b6* variable. However, for the rows there are no independent totals for *b4*, which is the nested variable there. In addition, there are no grand totals:



```
-----------------------------------------------------------------------------------------------
           |                 Civil status and Urban/Rural location of household
State and  | -- Never married --   - Married - monog -   - Married - polyg -   ------ Total ------
Sex        | Urban  Rural  Total   Urban  Rural  Total   Urban  Rural  Total   Urban  Rural  Total
-----------+-----------------------------------------------------------------------------------
02 Capital |
      Male |     1      .      1       2      .      2       2      .      2       5      .      5
    Female |     3      .      3       3      .      3       .      .      .       6      .      6
           |
     Total |     4      .      4       5      .      5       2      .      2      11      .     11
-----------+-----------------------------------------------------------------------------------
04 East    |
      Male |     2      .      2       1      .      1       .      .      .       3      .      3
    Female |     2      .      2       1      .      1       2      .      2       5      .      5
           |
     Total |     4      .      4       2      .      2       2      .      2       8      .      8
-----------+-----------------------------------------------------------------------------------
05 South   |
      Male |     .      1      1       .      1      1       .      .      .       .      2      2
    Female |     .      2      2       .      2      2       .      .      .       .      4      4
           |
     Total |     .      3      3       .      3      3       .      .      .       .      6      6
-----------+-----------------------------------------------------------------------------------
06 West    |
      Male |     5      .      5       1      .      1       .      .      .       6      .      6
    Female |     2      .      2       2      .      2       .      .      .       4      .      4
           |
     Total |     7      .      7       3      .      3       .      .      .      10      .     10
-----------------------------------------------------------------------------------------------
```

Missing values in the categories are always excluded. If we want to include missing values, we must recode to a valid value. We also change the value labels:

```
replace b6 = 9 if b6 == .

label define civil_status ///
1 "Never married" ///
2 "Married - monogamy" ///
3 "Married - polygamy" ///
4 "Widowed" ///
5 "Separated" ///
6 "Divorced" ///
9 "N/A", replace

table b4 urbrur b6, by(state) row col sc missing
```

The new table include the missing variables:

```
-----------------------------------------------------------------------------------------------------------
          |                  Civil status and Urban/Rural location of household
State and | — Never married — – Married - monog – – Married - polyg – ——— N/A ———     ——— Total ———
Sex       | Urban  Rural  Total  Urban  Rural  Total  Urban  Rural  Total  Urban  Rural  Total  Urban  Rural  Total
----------+-------------------------------------------------------------------------------------------------
02 Capital|
     Male |     1      .      1      2      .      2      2      .      2      2      .      2      7      .      7
   Female |     3      .      3      3      .      3      .      .      .      4      .      4     10      .     10
          |
    Total |     4      .      4      5      .      5      2      .      2      6      .      6     17      .     17
----------+-------------------------------------------------------------------------------------------------
04 East   |
     Male |     2      .      2      1      .      1      .      .      .      1      .      1      4      .      4
   Female |     2      .      2      1      .      1      2      .      2      3      .      3      8      .      8
          |
    Total |     4      .      4      2      .      2      2      .      2      4      .      4     12      .     12
----------+-------------------------------------------------------------------------------------------------
05 South  |
     Male |     .      1      1      .      1      1      .      .      .      .      .      .      .      2      2
   Female |     .      2      2      .      2      2      .      .      .      .      .      .      .      4      4
          |
    Total |     .      3      3      .      3      3      .      .      .      .      .      .      .      6      6
----------+-------------------------------------------------------------------------------------------------
06 West   |
     Male |     5      .      5      1      .      1      .      .      .      2      .      2      8      .      8
   Female |     2      .      2      2      .      2      .      .      .      .      .      .      4      .      4
          |
    Total |     7      .      7      3      .      3      .      .      .      2      .      2     12      .     12
-----------------------------------------------------------------------------------------------------------
```

When it comes to tables with stacked variables, it is not possible within these tabulation procedures. Instead, we can divide the table into separate two-way tables:

```
tab2 state b6, missing
tab2 state urbrur
tab2 b4 b6, missing
tab2 b4 urbrur
```

There are some problems with the labels for the *b6* variable, they are truncated. In the *table* procedure it is possible to set the cell width up to 20 positions, however that is not possible in the *tab2* procedure. The 4 tables:

```
           |                  Civil status
     State | Never mar  Married -  Married -        N/A |     Total
-----------+--------------------------------------------+----------
02 Capital |         4          5          2          6 |        17
   04 East |         4          2          2          4 |        12
  05 South |         3          3          0          0 |         6
   06 West |         7          3          0          2 |        12
-----------+--------------------------------------------+----------
     Total |        18         13          4         12 |        47
           | Urban/Rural location
           |     of household
     State |     Urban      Rural |     Total
-----------+----------------------+----------
02 Capital |        17          0 |        17
   04 East |        12          0 |        12
  05 South |         0          6 |         6
   06 West |        12          0 |        12
-----------+----------------------+----------
     Total |        41          6 |        47
```

```
           |                 Civil status
       Sex | Never mar  Married -  Married -        N/A |     Total
-----------+--------------------------------------------+----------
      Male |         9          5          2          5 |        21 
    Female |         9          8          2          7 |        26 
-----------+--------------------------------------------+----------
     Total |        18         13          4         12 |        47 

           | Urban/Rural location
           |    of household
       Sex |     Urban      Rural |     Total
-----------+----------------------+----------
      Male |        19          2 |        21 
    Female |        22          4 |        26 
-----------+----------------------+----------
     Total |        41          6 |        47 
```

Now we want to calculate the average age by *state* and *b4*. We use the contents option with the mean statistic and the variable name, *b5*. We format the cells with a fixed format with no decimals.

```
table state b4, contents(mean b5) row col format(%9.0f)
```

The table cells are now average age without decimals:

```
-----------------------------------
           |          Sex          
     State |   Male  Female   Total
-----------+-----------------------
02 Capital |     24      16      19
   04 East |     18      17      18
  05 South |     40      32      34
   06 West |     18      28      22
           | 
     Total |     22      21      21
-----------------------------------
```

When we want to make frequency tables, we turn back to the *tab2* procedure. We add the *cell* option to add the percentages of the grand total and *nofreq* to suppress the frequencies to the first table. Then we use the *row* option to have row percentages in the second and *col* for column percentages in the third table:

```
tab2 state b4, cell nofreq
tab2 state b4, row nofreq
tab2 state b4, col nofreq
```

We have now three tables with percentages of the grand total, the rows and the columns:

```
           |         Sex
     State |     Male     Female |     Total
-----------+----------------------+----------
02 Capital |    14.89      21.28 |     36.17
   04 East |     8.51      17.02 |     25.53
  05 South |     4.26       8.51 |     12.77
   06 West |    17.02       8.51 |     25.53
-----------+----------------------+----------
     Total |    44.68      55.32 |    100.00
```

```
           |         Sex
     State |     Male     Female |     Total
-----------+----------------------+----------
02 Capital |    41.18      58.82 |    100.00
   04 East |    33.33      66.67 |    100.00
  05 South |    33.33      66.67 |    100.00
   06 West |    66.67      33.33 |    100.00
-----------+----------------------+----------
     Total |    44.68      55.32 |    100.00
```

```
           |         Sex
     State |     Male     Female |     Total
-----------+----------------------+----------
02 Capital |    33.33      38.46 |     36.17
   04 East |    19.05      30.77 |     25.53
  05 South |     9.52      15.38 |     12.77
   06 West |    38.10      15.38 |     25.53
-----------+----------------------+----------
     Total |   100.00     100.00 |    100.00
```

Finally we wil combine the absolute figures with percentages. To do that we just delete the *nofreq* option:

`tab2 state b4, col`

The absolute figures and the percentages will be placed in the different rows, not in different columns:

```
           |         Sex
     State |     Male     Female |     Total
-----------+----------------------+----------
02 Capital |        7         10 |        17
           |    33.33      38.46 |     36.17
-----------+----------------------+----------
   04 East |        4          8 |        12
           |    19.05      30.77 |     25.53
-----------+----------------------+----------
  05 South |        2          4 |         6
           |     9.52      15.38 |     12.77
-----------+----------------------+----------
   06 West |        8          4 |        12
           |    38.10      15.38 |     25.53
-----------+----------------------+----------
     Total |       21         26 |        47
           |   100.00     100.00 |    100.00
```

These tabulation procedures have very concise syntax and that makes the possibilities somewhat limited. The additional *tabout* procedure (see https://www.ianwatson.com.au/stata/tabout_tutorial.pdf) is also a possibility, however its syntax is rather complex.

21.4.R

There are no flexible tabulation commands in base R. However, there is a package called *tables* that includes the function *tabular*. This is a flexible function that can create many kinds of tables. It seems to be inspired by the *Proc tabulate* procedure in Sas, as the syntax is similar to the *table* statement in *Proc tabulate*.

We activate the tabular procedure with the library function, if it is not installed we have to do that first with the *install.packages* command:

```
install.packages("tables")
library(tables)
```

The dimensions are separated by the ~ sign. To nest variables and to combine with measures, we use the * sign. We can stack variables with the + sign. To add totals and subtotals we simply add a 1 where we want them. Our categorical variables must be defined as factor variables.

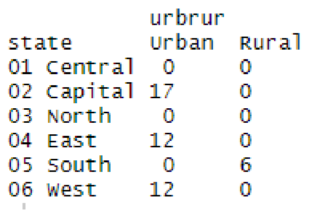
The output can be displayed directly in the Console or written to an R object. When it is written to an object it consists of several lists. The output in the Console window is a plain text and it is not easy to export it properly to other formats. However, we may wrap the table into other functions, like *knitr* or *latex* to convert to for instance html.

There is no option within the tabular function to add titles to the table.

First, we will look at a simple two-way table with *state* in the rows and *urbrur* in the columns:

```
tabular(state ~ urbrur,data=mdgperson_nodup)
```

The table:

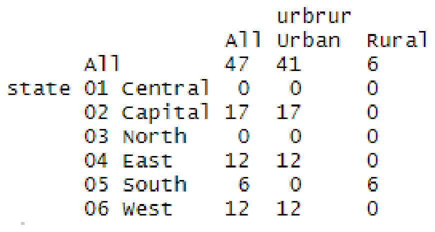


```
           urbrur
 state     Urban  Rural
 01 Central  0     0
 02 Capital 17     0
 03 North    0     0
 04 East    12     0
 05 South    0     6
 06 West    12     0
```

We want to add totals and do that with adding 1 before the variables because we want the totals before the distributions. The + sign is used to stack the total with the categorical variables:

```
tabular(1 + state ~ 1 + urbrur,data=mdgperson_nodup)
```

Now totals are included:



```
                       urbrur
                   All Urban  Rural
      All          47  41     6
state 01 Central    0   0     0
      02 Capital   17  17     0
      03 North      0   0     0
      04 East      12  12     0
      05 South      6   0     6
      06 West      12  12     0
```

To nest variables in the rows we use the * sign:

```
tabular(1 + state*b4 ~ 1 + urbrur,data=mdgperson_nodup)
```

Here we see that *b4* is distributed for every value of *state*:

```
                              urbrur
                          All Urban  Rural
                    All    47 41     6
state 01 Central b4 Male    0  0     0
                    Female  0  0     0
      02 Capital    Male    7  7     0
                    Female 10 10     0
      03 North      Male    0  0     0
                    Female  0  0     0
      04 East       Male    4  4     0
                    Female  8  8     0
      05 South      Male    2  0     2
                    Female  4  0     4
      06 West       Male    8  8     0
                    Female  4  4     0
```

As mentioned, categorical variables must be defined as factors. To include NA's we have to change *b6* to a factor which don't exclude them before we can use it in our table. We can now nest *b6* with *urbrur*:

```
mdgperson_nodup$b6 <- factor(mdgperson_nodup$b6, exclude = NULL)
tabular(1 + state*b4 ~ 1 + b6*urbrur,data=mdgperson_nodup)
```

The table will now include the NA's:

```
                              b6
                              Never married       Married - monogamy  Married - polygamy  NA
                              urbrur              urbrur              urbrur              urbrur
                          All Urban        Rural  Urban        Rural  Urban        Rural  Urban        Rural
                    All    47 15           3      10           3      4            0      12           0
state 01 Central b4 Male    0 0            0      0            0      0            0      0            0
                    Female  0 0            0      0            0      0            0      0            0
      02 Capital    Male    7 1            0      2            0      2            0      2            0
                    Female 10 3            0      3            0      0            0      4            0
      03 North      Male    0 0            0      0            0      0            0      0            0
                    Female  0 0            0      0            0      0            0      0            0
      04 East       Male    4 2            0      1            0      0            0      1            0
                    Female  8 2            0      1            0      2            0      3            0
      05 South      Male    2 0            1      0            1      0            0      0            0
                    Female  4 0            2      0            2      0            0      0            0
      06 West       Male    8 5            0      1            0      0            0      2            0
                    Female  4 2            0      2            0      0            0      0            0
```

We will add subtotals and also have *b4* and *urbrur* distributed on the totals. We do that by adding parenthesis. The expression (1 + state)*(1 + b4) will combine 1 and state to both 1 and b4. 1*1 will be the grand total and 1 * b4 the subtotals. We also add headings and they must be placed before the variable and be connected with the * sign:

```
tabular((1 + state)*(Heading('Both sexes')*1+b4) ~ (1 + b6)*(Heading('Both
locations')*1 + urbrur),data=mdgperson_nodup)
```

The table with the grand total and all the subtotals:

```
                                                          b6
                              All                         Never married               Married - monogamy          Married - polygamy          NA
                                             urbrur                      urbrur                      urbrur                      urbrur                      urbrur
                              Both locations Urban  Rural Both locations Urban  Rural Both locations Urban  Rural Both locations Urban  Rural Both locations Urban  Rural
     All           Both sexes 47             41     6     18             15     3     13             10     3     4              4      0     12             12     0
                b4 Male       21             19     2     9              8      1     5              4      1     2              2      0     5              5      0
                   Female     26             22     4     9              7      2     8              6      2     2              2      0     7              7      0
state 01 Central   Both sexes 0              0      0     0              0      0     0              0      0     0              0      0     0              0      0
                b4 Male       0              0      0     0              0      0     0              0      0     0              0      0     0              0      0
                   Female     0              0      0     0              0      0     0              0      0     0              0      0     0              0      0
      02 Capital   Both sexes 17             17     0     4              4      0     5              5      0     2              2      0     6              6      0
                b4 Male       7              7      0     1              1      0     2              2      0     2              2      0     2              2      0
                   Female     10             10     0     3              3      0     3              3      0     0              0      0     4              4      0
      03 North     Both sexes 0              0      0     0              0      0     0              0      0     0              0      0     0              0      0
                b4 Male       0              0      0     0              0      0     0              0      0     0              0      0     0              0      0
                   Female     0              0      0     0              0      0     0              0      0     0              0      0     0              0      0
      04 East      Both sexes 12             12     0     4              4      0     2              2      0     2              2      0     4              4      0
                b4 Male       4              4      0     2              2      0     1              1      0     0              0      0     1              1      0
                   Female     8              8      0     2              2      0     1              1      0     2              2      0     3              3      0
      05 South     Both sexes 6              0      6     3              0      3     3              0      3     0              0      0     0              0      0
                b4 Male       2              0      2     1              0      1     1              0      1     0              0      0     0              0      0
                   Female     4              0      4     2              0      2     2              0      2     0              0      0     0              0      0
      06 West      Both sexes 12             12     0     7              7      0     3              3      0     0              0      0     2              2      0
                b4 Male       8              8      0     5              5      0     1              1      0     0              0      0     2              2      0
                   Female     4              4      0     2              2      0     2              2      0     0              0      0     0              0      0
```

For stacking variables, we use the + sign:

```
tabular(1 + state + b4 ~ 1 + b6 + urbrur,data=mdgperson_nodup)
```

This is the table with stacked variables:

```
                   b6                                                    urbrur
                All Never married Married - monogamy Married - polygamy NA Urban  Rural
      All       47  18            13                 4                  12 41     6
state 01 Central  0   0             0                 0                  0  0     0
      02 Capital 17   4             5                 2                  6 17     0
      03 North    0   0             0                 0                  0  0     0
      04 East    12   4             2                 2                  4 12     0
      05 South    6   3             3                 0                  0  0     6
      06 West    12   7             3                 0                  2 12     0
b4    Male       21   9             5                 2                  5 19     2
      Female     26   9             8                 2                  7 22     4
```

When we want to add measures, the default is that they are not calculated when there are NA's (missing values). That means the table will be filled with NA's in these cells. We see that when we introduce the b5 variable to calculate the average ages. We have also formatted the numbers in the cells with no decimals:

```
tabular(1 + state ~ (1 +
Heading()*b4)*Heading()*b5*Heading()*mean*Format(digits=1),data=mdgperson_nod
up)
```

The table looks like this:

```
                All Male Female
      All        NA  NA   21
state 01 Central NaN NaN  NaN
      02 Capital  19  24   16
      03 North   NaN NaN  NaN
      04 East     18  18   17
      05 South    34  40   32
      06 West     NA  NA   28
```

To calculate average values for all non-missing values, we can create a new *Mean* function that will exclude NA's and the create the table again:

```
Mean <- function(x) base::mean(x, na.rm=TRUE)
tabular(1 + state ~ (1 +
Heading()*b4)*Heading()*b5*Heading()*Mean*Format(digits=1),data=mdgperson_nod
up)
```

Now only those where all values are missing will be NA:

```
                All Male Female
      All        21  22   21
state 01 Central NaN NaN  NaN
      02 Capital  19  24   16
      03 North   NaN NaN  NaN
      04 East     18  18   17
      05 South    34  40   32
      06 West     22  18   28
```

If we want to exclude the empty categories for *state* we can exclude the missing values:

```
mdgperson_nodup$state <- factor(mdgperson_nodup$state, exclude = NULL)
tabular(1 + state ~ (1 +
Heading()*b4)*Heading()*b5*Heading()*Mean,data=mdgperson_nodup)
```

The new table omits the empty categories:

```
                   All Male Female
           All        21   22    21
    state 02 Capital 19   24    16
          04 East     18   18    17
          05 South    34   40    32
          06 West     22   18    28
```

Now we will look at percentages, both calculated on the grand total, rows and columns. We use the *Percent* option, first without any parameters for the grand total. Then *row* and *col* parameters for row and column percentages:

```
tabular(1 + state ~ (1 + b4)*Percent()*Format(digits=1),data=mdgperson_nodup)
tabular(1 + state ~ (1 +
b4)*(RowPct=Percent("row"))*Format(digits=1),data=mdgperson_nodup)
tabular(1 + state ~ (1 +
b4)*(ColPct=Percent("col"))*Format(digits=1),data=mdgperson_nodup)
```

The percentage tables:

```
                          b4                                          b4
                  All     Male    Female                   All    Male   Female
                  Percent Percent Percent                  RowPct RowPct RowPct
         All      100      45      55             All      100     45     55
   state 02 Capital 36     15      21       state 02 Capital 100   41     59
         04 East   26       9      17             04 East  100     33     67
         05 South  13       4       9             05 South 100     33     33
         06 West   26      17       9             06 West  100     67     33
                          b4
                   All    Male   Female
                   ColPct ColPct ColPct
          All      100    100    100
    state 02 Capital 36    33     38
          04 East    26    19     31
          05 South   13    10     15
          06 West    26    38     15
```

Finally, we will combine absolute figures with column percentages:

```
tabular(Heading('Total')*1 + state ~ Heading('N')*(Heading('Total')*1 +
Heading()* b4) + (Heading('%')*(Heading('Total')*1 + Heading()* b4)*
Heading()* Percent("col"))*Format(digits=1),data=mdgperson_nodup)
```

The table now includes both absolute figures and column percentages:

```
                   N                    %
                   Total Male Female Total Male Female
          Total    47    21   26     100   100  100
    state 02 Capital 17   7   10      36    33   38
          04 East  12     4    8      26    19   31
          05 South  6     2    4      13    10   15
          06 West  12     8    4      26    38   15
```

The *tabular* function is a very flexible tabulation tool. However, the output format has by default a very simple layout.

21.5. Python

In Python, we have the Pandas crosstab we can use for tabulation. It has a simple syntax and can create tables with more nested levels both in the rows and columns. It does not stack variables in either dimension and it does not support sub-totals.

We start with a simple two-way table. The row variable is defined first, then the column variable:

```
pd.crosstab(mdgperson_nodup.state, columns=mdgperson_nodup.urbrur)
```

The output table:

urbrur	Rural	Urban
state		
02 Capital	0	17
04 East	0	12
05 South	6	0
06 West	0	12

We will now add totals and also name them. Furthermore, we give name to the row variable and suppress the column name:

```
pd.crosstab(mdgperson_nodup.state, columns=mdgperson_nodup.urbrur,
margins=True, margins_name='Total',
                    rownames=['State'], colnames=['']
                  )
```

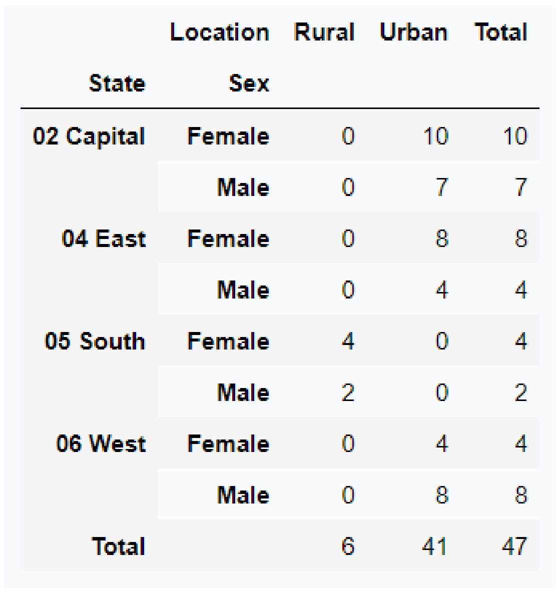
The output table with totals, they are always placed at the end:

	Rural	Urban	Total
State			
02 Capital	0	17	17
04 East	0	12	12
05 South	6	0	6
06 West	0	12	12
Total	6	41	47

Now we will add sex to the rows as a sub-item to state. To do that we add the b4 variable and put it in brackets together with state. We also add Sex to the rownames in a similar way:

```
pd.crosstab([mdgperson_nodup.state, mdgperson_nodup.b4],
columns=mdgperson_nodup.urbrur, margins=True, margins_name='Total',
                    rownames=['State', 'Sex'], colnames=['Location']
                  )
```

There are no subtotals within each state in the output table:

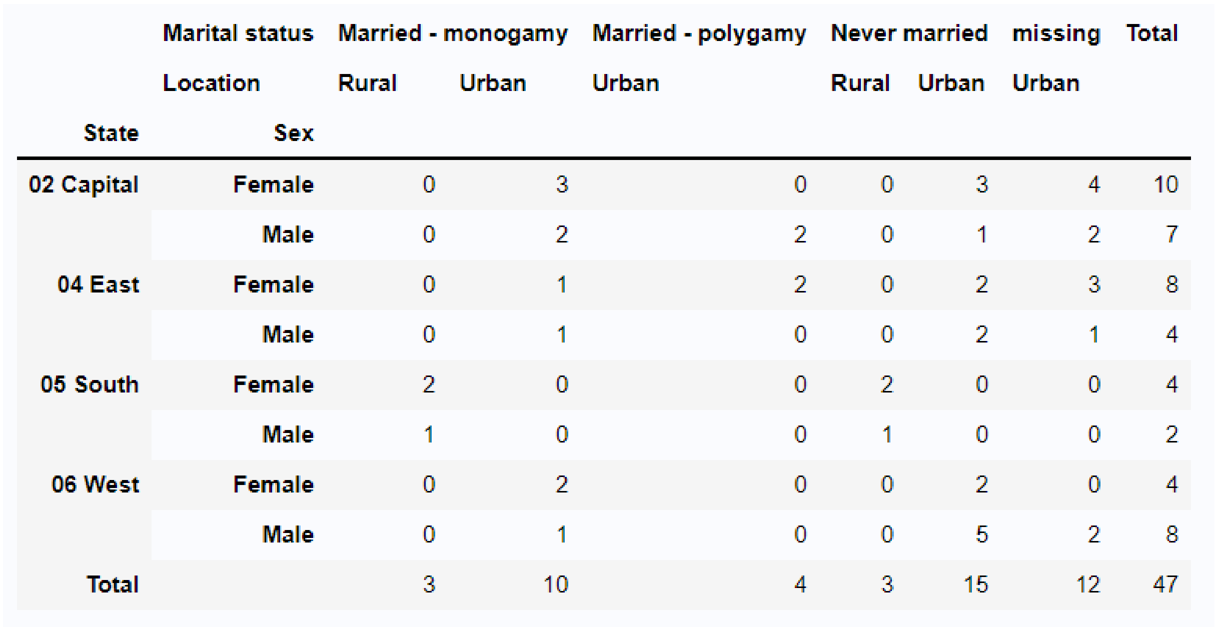


	Location	Rural	Urban	Total
State	**Sex**			
02 Capital	**Female**	0	10	10
	Male	0	7	7
04 East	**Female**	0	8	8
	Male	0	4	4
05 South	**Female**	4	0	4
	Male	2	0	2
06 West	**Female**	0	4	4
	Male	0	8	8
Total		6	41	47

We can have nested variables in the columns as well, we just add the variable to the columns and colnames parameters:

```
pd.crosstab([mdgperson_nodup.state, mdgperson_nodup.b4],
columns=[mdgperson_nodup.b6, mdgperson_nodup.urbrur],
            margins=True, margins_name='Total', rownames=['State', 'Sex'],
colnames=['Marital status', 'Location']
           )
```

We have now a table with two nested variables both in the rows and columns. We see that there are no columns for the combinations of Marital status and Location that is not in the data:



	Marital status	Married - monogamy		Married - polygamy	Never married		missing	Total
	Location	**Rural**	**Urban**	**Urban**	**Rural**	**Urban**	**Urban**	
State	**Sex**							
02 Capital	**Female**	0	3	0	0	3	4	10
	Male	0	2	2	0	1	2	7
04 East	**Female**	0	1	2	0	2	3	8
	Male	0	1	0	0	2	1	4
05 South	**Female**	2	0	0	2	0	0	4
	Male	1	0	0	1	0	0	2
06 West	**Female**	0	2	0	0	2	0	4
	Male	0	1	0	0	5	2	8
Total		3	10	4	3	15	12	47

Now we can look at how to add average values of a measure variable to our table. We only use on variable in each dimension in this example. The variable to calculate the averages by is added to the *values* parameter and we tell that we want the averages with the *aggfunc* parameter. We also style the figures with one decimal and decimal comma:

```
pd.crosstab(mdgperson_nodup.state, columns=mdgperson_nodup.b4,
values=mdgperson_nodup.b5, aggfunc='mean',
            margins=True, margins_name='Total'
           ).style.format(decimal=',', precision=1)
```

Missing values are omitted from the calculations:

b4	Female	Male	Total
state			
02 Capital	15,8	23,7	19,1
04 East	17,2	18,0	17,5
05 South	31,5	40,0	34,3
06 West	28,0	17,9	21,5
Total	20,5	22,1	21,2

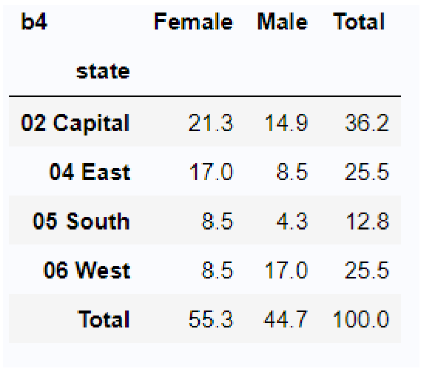
As seen above, we can do formatting with style.format. Here are some useful options:

• precision	Number of decimals
• decimal	Decimal sign
• thousands	Thousand separator
• na_rep	Representation of missing values

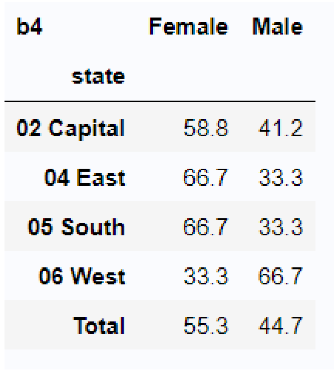
There are 3 basic percentage tables, percentages based on the grand total, the row total, and the column total. Here are examples on these 3. We choose which percentage calculation we want in the normalize parameter. We also use the *round* function and multiply with 100 to display the percentages with one decimal:

```
pd.crosstab(mdgperson_nodup.state, columns=mdgperson_nodup.b4, margins=True,
margins_name='Total',
            normalize=True
           ).round(3)*100

pd.crosstab(mdgperson_nodup.state, columns=mdgperson_nodup.b4, margins=True,
margins_name='Total',
            normalize='index'
           ).round(3)*100

pd.crosstab(mdgperson_nodup.state, columns=mdgperson_nodup.b4, margins=True,
margins_name='Total',
            normalize='columns'
           ).round(3)*100
```

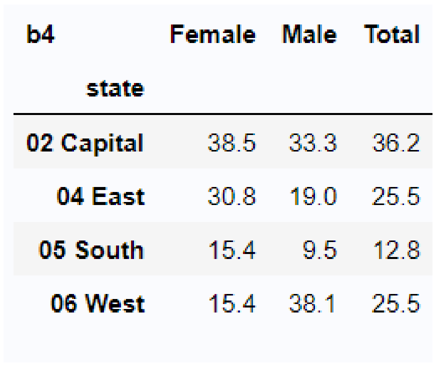
We see that the total column is omitted from the row percentages table and the total row is omitted from the column percentages table as the values are always 100:



b4	Female	Male	Total
state			
02 Capital	21.3	14.9	36.2
04 East	17.0	8.5	25.5
05 South	8.5	4.3	12.8
06 West	8.5	17.0	25.5
Total	55.3	44.7	100.0



b4	Female	Male
state		
02 Capital	58.8	41.2
04 East	66.7	33.3
05 South	66.7	33.3
06 West	33.3	66.7
Total	55.3	44.7



b4	Female	Male	Total
state			
02 Capital	38.5	33.3	36.2
04 East	30.8	19.0	25.5
05 South	15.4	9.5	12.8
06 West	15.4	38.1	25.5

Crosstab is easy to learn, however the tables we can create are limited.

22. Graphs

All these softwares have possibilities for creating different kind of graphs, like line plots, bar charts, pie charts, scatter plots and so on. There are lots of options when it comes to layout and combining different graphs in the same chart. We will now have a brief glimpse of some bar charts just to lead the way to different and more complex graphs.

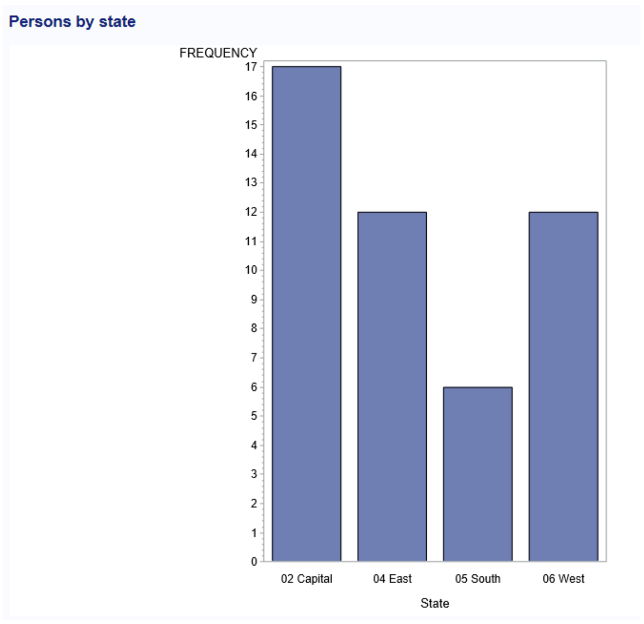
22.1. Sas

There is an additional module with extra license fee for graphs in Sas, called Sas/Graph. When it is available, we can create many different graphs and also enhance the layout. There are some basic chart procedures with limited layout like Proc chart and Proc plot. Sas have made more flexible procedures called Proc gchart and Proc gplot, which we will use here.

We start with a simple vertical bar chart which counts the number of persons (observations) by state. There will we one bar for each state. We use the *discrete* option to make sure that there will be one bar for each state:

```
proc gchart data=mdg.mdgperson_nodup;
 vbar state /discrete;
 title 'Persons by state';
run;
```

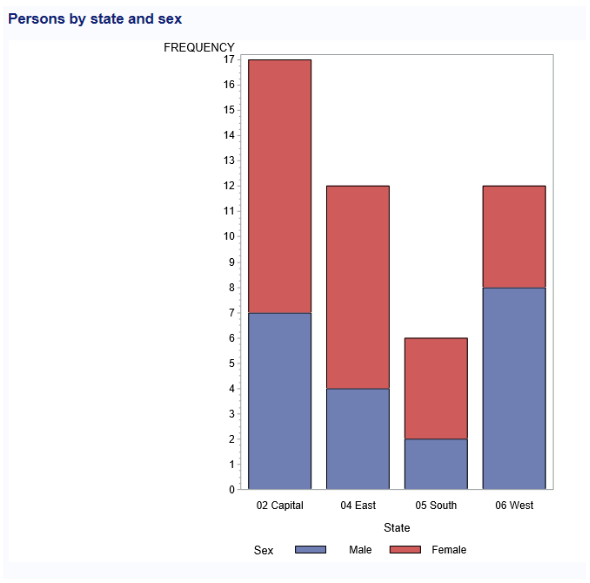
This syntax will create a graph like this:



We can introduce another variable, *b4*, to create a stacked bar chart. When we define it as a *subgroup*, it will be stacked:

```
proc gchart data=mdg.mdgperson_nodup;
 vbar state / subgroup=b4 discrete;
 title 'Persons by state and sex';
run;
```

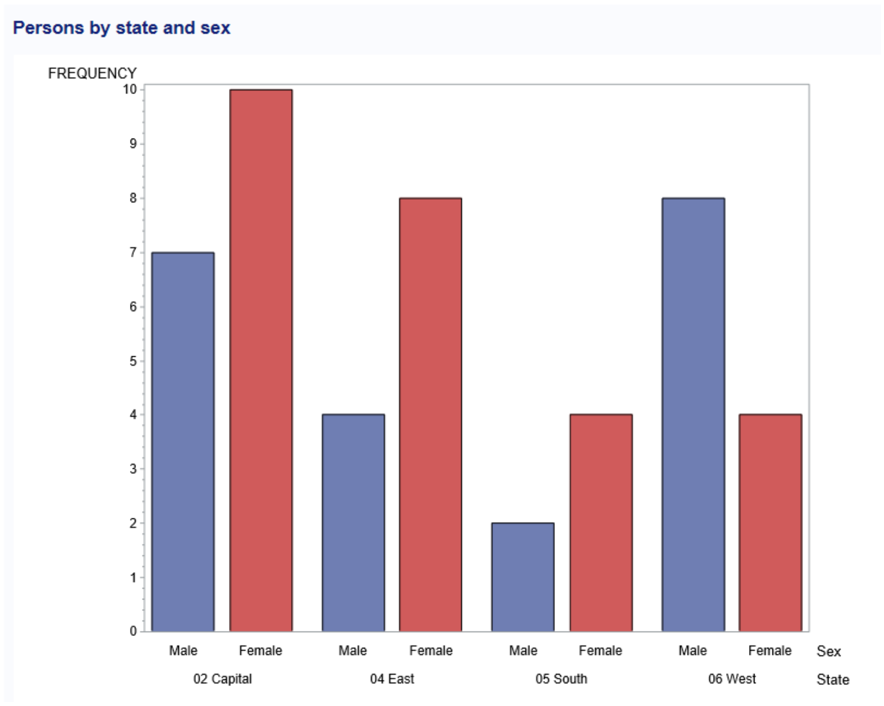
We see that we have sex stacked in each bar:



Now we can put the sex variable in separate columns. To do that, we can change the order of the variables in the *vbar* statement and make *state* a group instead of a subgroup. To get the same colour for the same sex, we use the parameter *midpoint* for the *patternid* option:

```
proc gchart data=mdg.mdgperson_nodup;
 vbar b4 / group=state discrete patternid=midpoint ;
 title 'Persons by state and sex';
run;
```

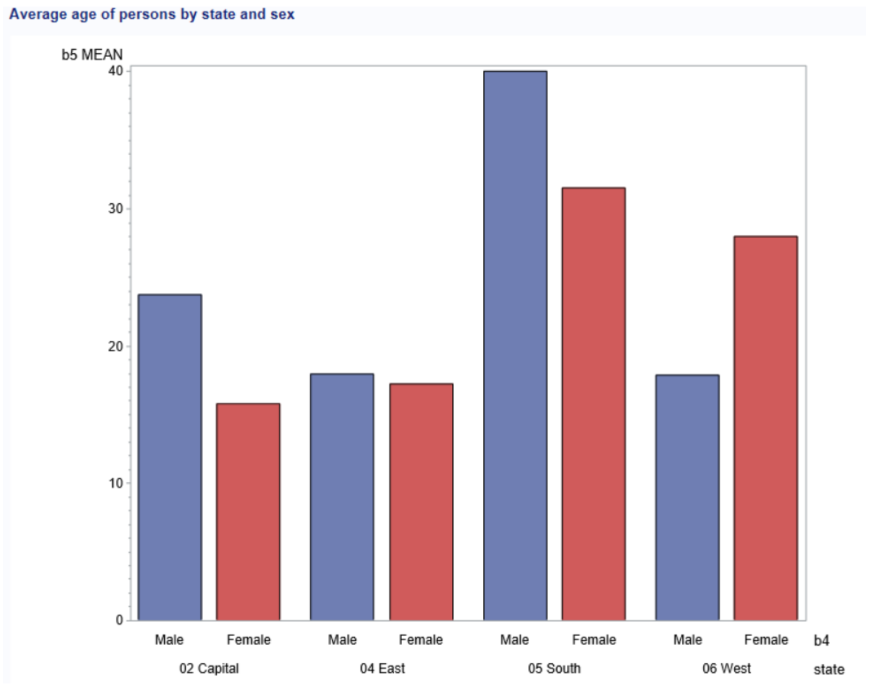
Sex is now in separate bars within each state:



We will now add the measure variable *b5* (age) and calculate the average age. To do that we use the *type* and *sumvar* options:

```
proc gchart data=mdg.mdgperson_nodup;
 vbar b4 / type=mean group=state sumvar=b5 discrete patternid=midpoint ;
 title 'Average age of persons by state and sex';
run;
```

The graph made for average age:

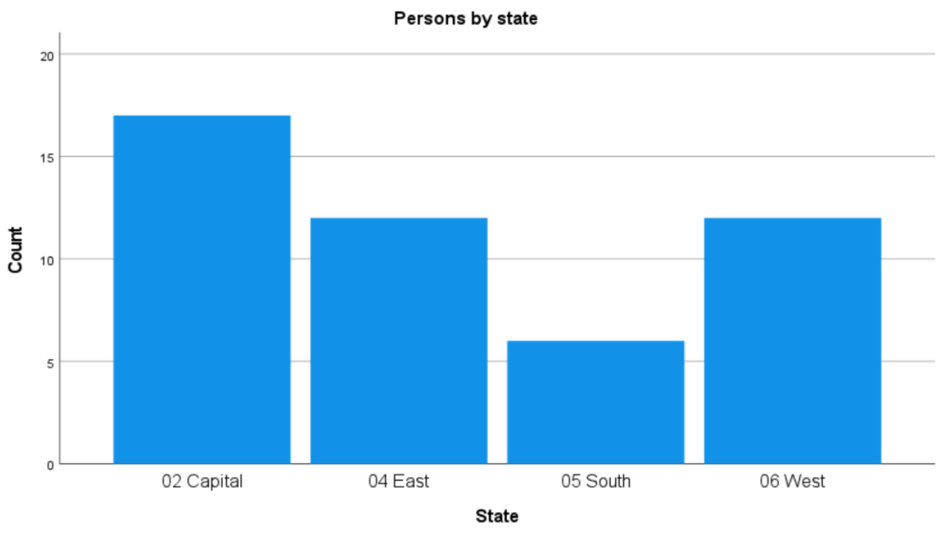


22.2. Spss

In Spss, we have a *Graph* command that creates graphs with nice layout. There is also the *Ggraph* command, which is more flexible. However, the syntax is much more complicated. The *Ggraph* also needs the GPL (Graphics Productions Language) to define the graph. We will look at some vertical bar charts. First, one with one bar for each state:

```
DATASET CLOSE all.
GET FILE='mdgperson_nodup.sav'.
GRAPH /BAR = count by state
      /TITLE='Persons by state'.
```

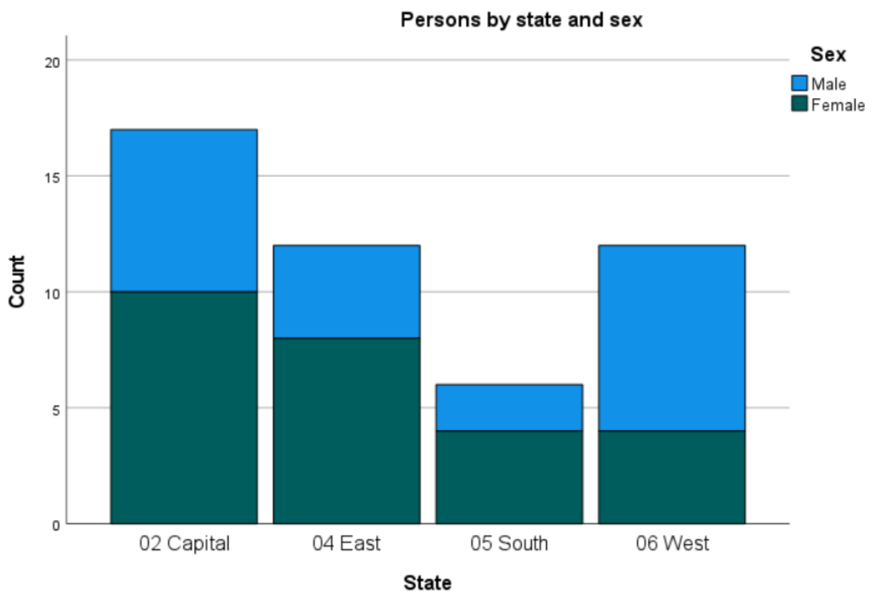
The syntax above will give us this graph:



We add the *b4* variable and stack it for each state:

```
GRAPH /BAR (stacked) = count by state b4
      /TITLE='Persons by state and sex'.
```

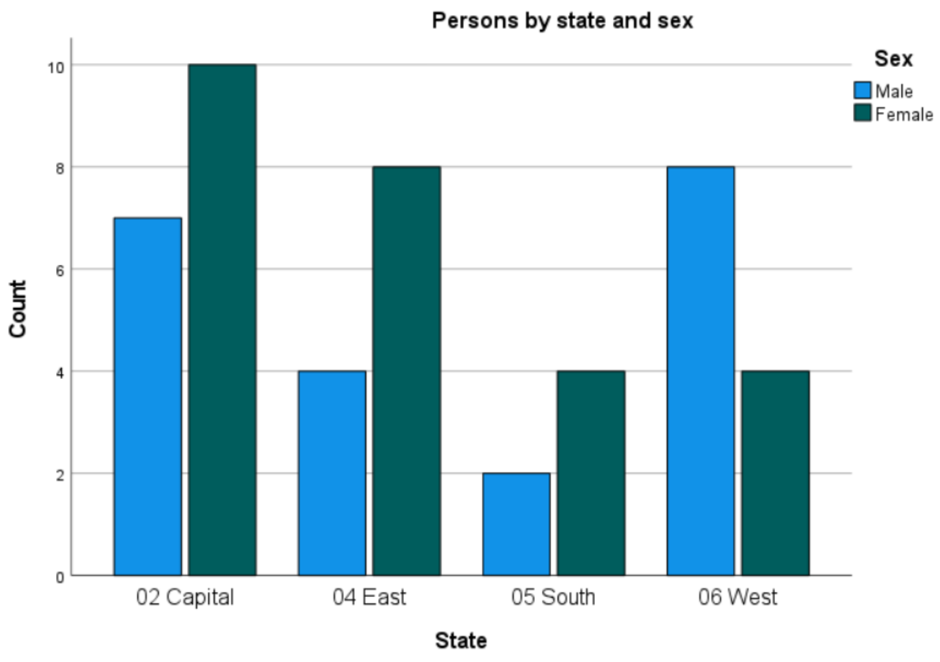
Now the bars show the distribution by sex:



We can put the sex variable into separate bars:

```
GRAPH /BAR = count by state b4
      /TITLE='Persons by state and sex'.
```

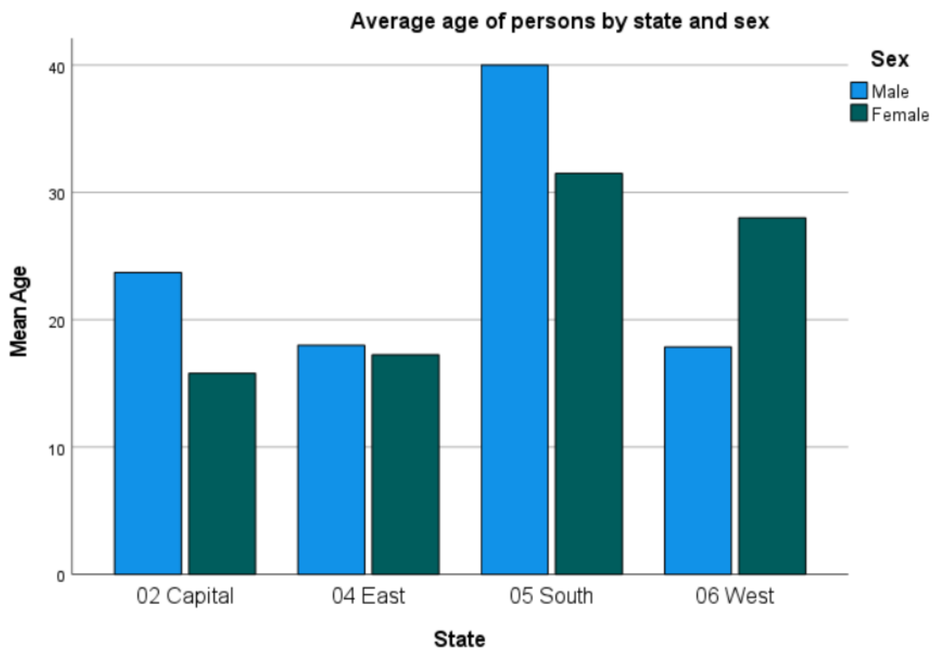
The graph:



Finally, we can add a measure variable, *b5*, to the graph and make the average age:

```
GRAPH /BAR = mean(b5) by state b4
      /TITLE='Average age of persons by state and sex'.
```

The graph looks like this:



We can use the menus in Spss to create graphs. However, the pasted syntax will be for the Ggraph command. We find the graphs under Graphs > Chart builder. Here is the generated syntax for the same graph as the first one above:

```
GGRAPH
  /GRAPHDATASET NAME="graphdataset" VARIABLES=state COUNT()[name="COUNT"]
MISSING=LISTWISE
    REPORTMISSING=NO
  /GRAPHSPEC SOURCE=INLINE.
BEGIN GPL
  SOURCE: s=userSource(id("graphdataset"))
  DATA: state=col(source(s), name("state"), unit.category())
  DATA: COUNT=col(source(s), name("COUNT"))
  GUIDE: axis(dim(1), label("State"))
  GUIDE: axis(dim(2), label("Count"))
  GUIDE: text.title(label("Persons by state"))
  SCALE: linear(dim(2), include(0))
  ELEMENT: interval(position(state*COUNT), shape.interior(shape.square))
END GPL.
```

As we see, the syntax for the Graph command is much easier than the Ggraph syntax. However, there are more possibilities when using Ggraph.

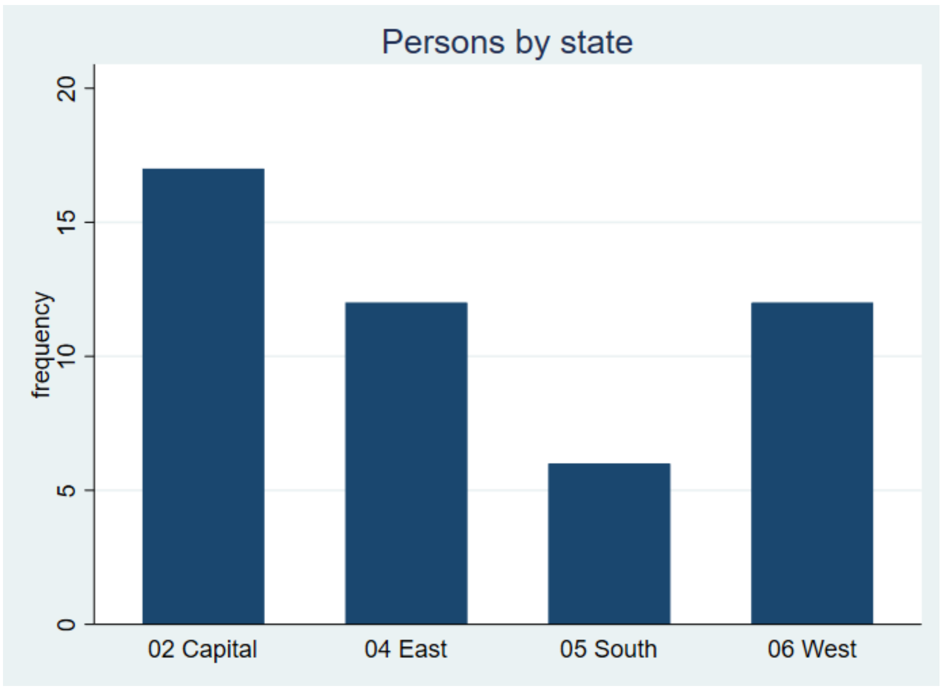
22.3. Stata

In Stata, we have separate commands for different plots. There is for instance the bar command for vertical bar charts, line for line plots and the graph pie command for pie charts.

We can make a simple vertical bar chart for the state variable like this:

```
use "mdgperson_nodup.dta", clear
graph bar (count), over(state) title(Persons by state)
```

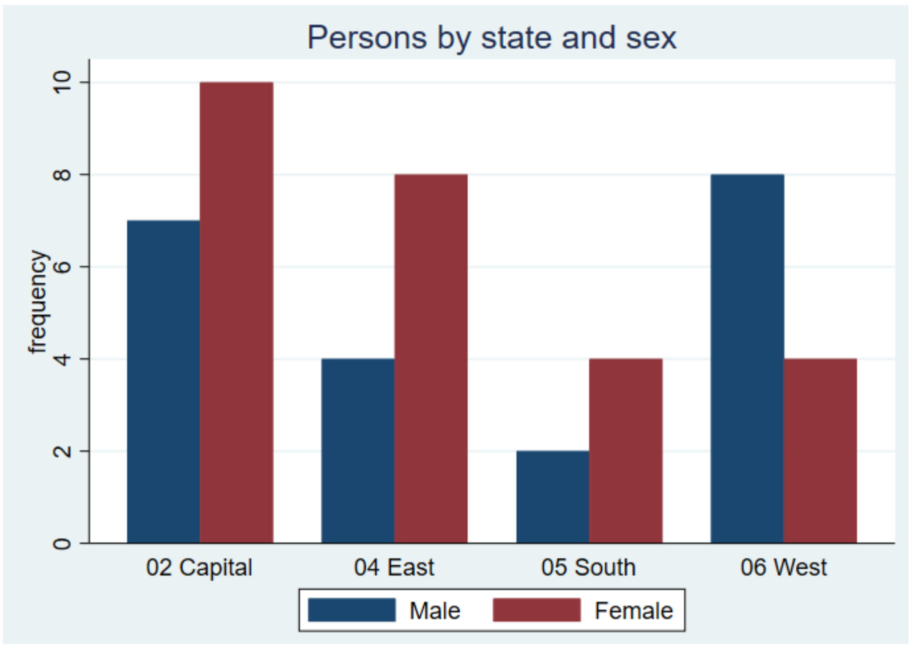
This will produce a graph like this:



We introduce the *b4* variable and make separate bars for each value:

```
graph bar (count), over(b4) over(state) title(Persons by state and sex) asyvars
```

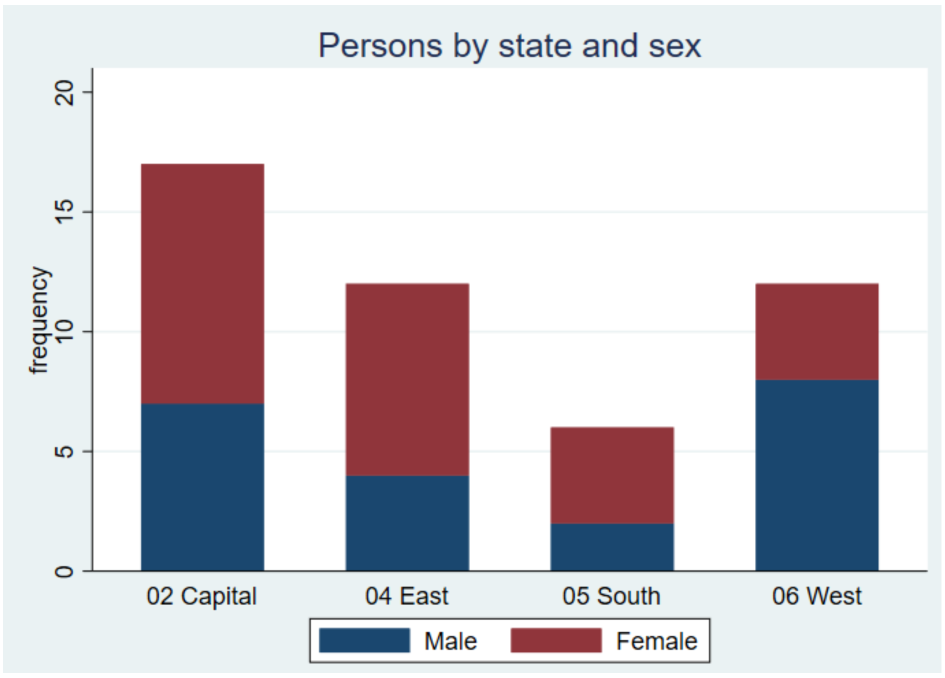
The *asyvars* option will give us different colours for the sexes:



We can use the *stack* option to put the sex variable in the same bar for each state:

```
graph bar (count), over(b4) over(state) stack title(Persons by state and sex) asyvars
```

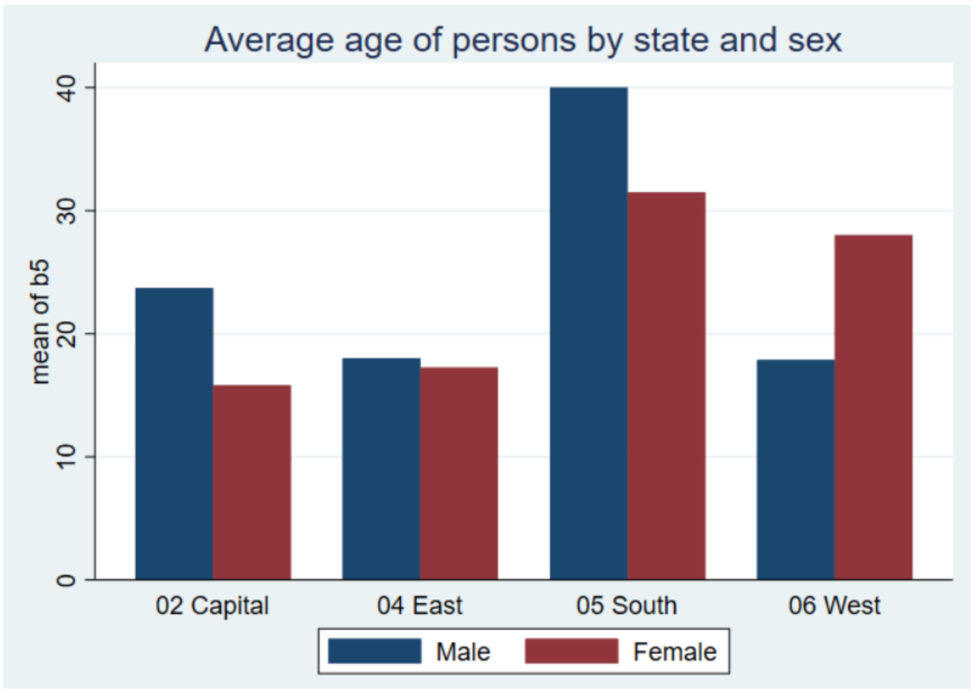
The graph has now stacked bars for each *state*:



We can add a measure variable to our graph, here we choose to calcluate the average age (*b5*) of the persons:

```
graph bar (mean) b5, over(b4) over(state) title(Average age of persons by state and
sex) asyvars
```

The graph:

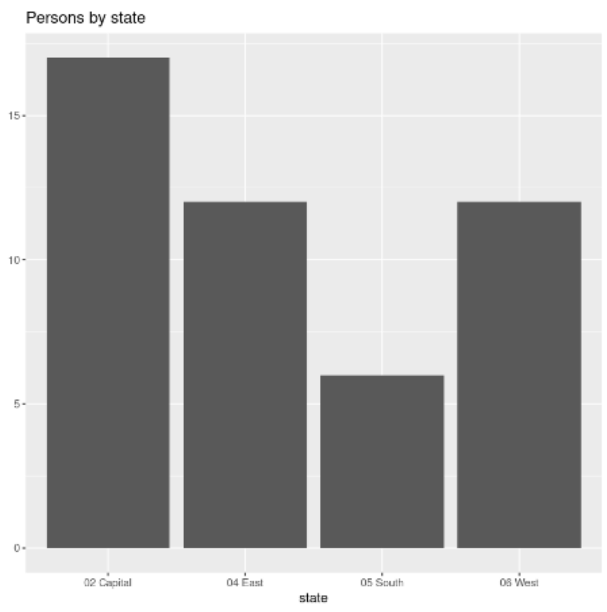


22.4. R

In R there are different procedures for creating graphs. One of the most common is ggplot2. It has a broad variety of graphs, see https://ggplot2.tidyverse.org/. Now we will create some bar charts. We start with a vertical bar chart that counts the number of persons (rows) for each *state*:

```
library(ggplot2)
mdgperson_nodup <- readRDS("../data/mdgperson_nodup.rds")
ggplot(data=mdgperson_nodup, aes(x=state)) +
  geom_bar(stat="count") +
  ggtitle("Persons by state")
```

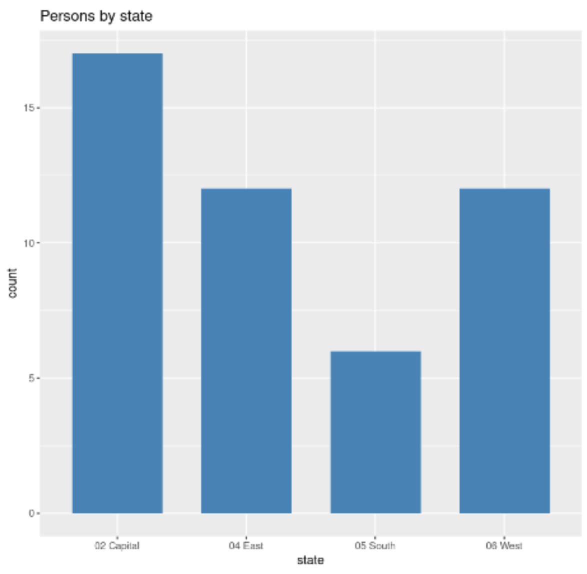
This will give us a very simple bar chart:



We can add some colours and make the bars a little bit slimmer:

```
ggplot(mdgperson_nodup, aes(x=state)) +
  geom_bar(stat="count", width=0.7, fill="steelblue") +
  ggtitle("Persons by state")
```

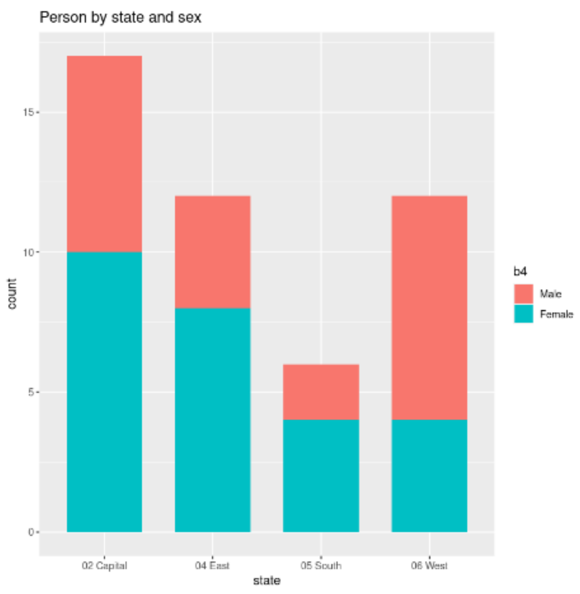
It looks better now:



Now we want to add the *b4* variable to our chart:

```
ggplot(mdgperson_nodup, aes(x=state, fill=b4)) +
  geom_bar(stat="count", width=0.7) +
  ggtitle("Persons by state and sex")
```

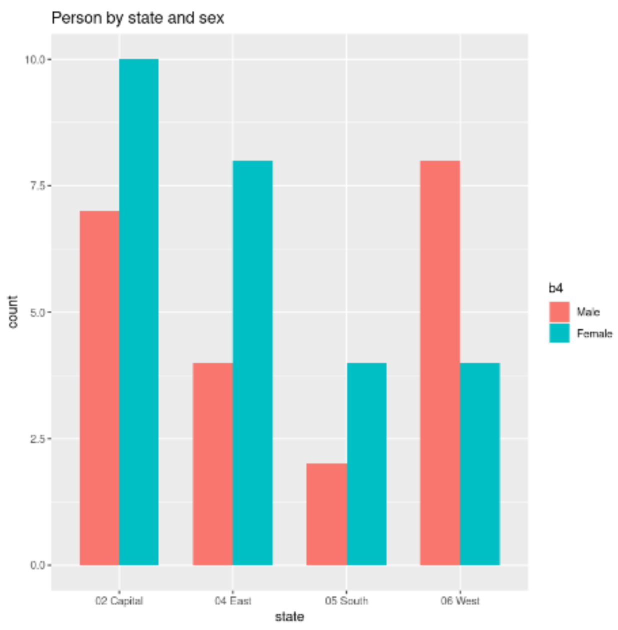
As we now have a fill variable, R chooses automatically different colours for each of the stacked sexes:



We can put sex in separate bars within each state by adding the *position* option:

```
ggplot(mdgperson_nodup, aes(x=state, fill=b4)) +
  geom_bar(stat="count", width=0.7, position=position_dodge())+
  ggtitle("Persons by state and sex")
```

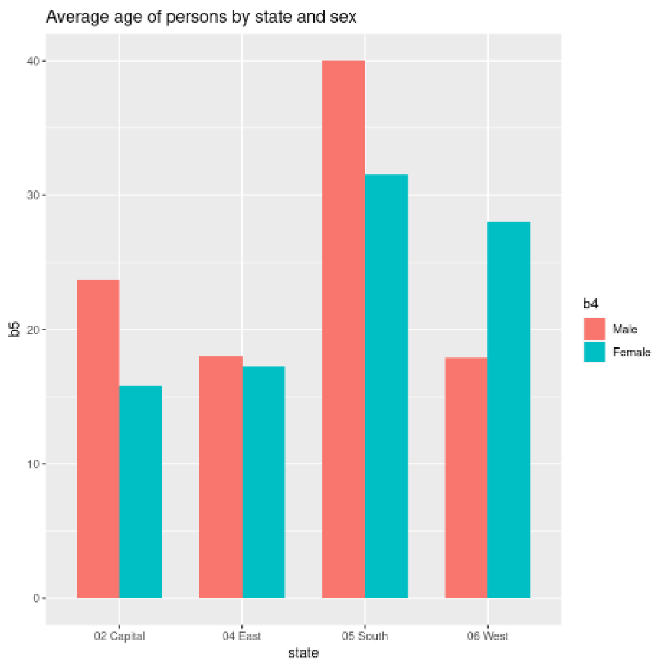
The graph:



We introduce a measure variable, for instance *b5* and calculate the mean values:

```
ggplot(mdgperson_nodup, aes(x=state, y=b5, fill=b4)) +
  geom_bar(stat="summary", fun="mean", width=0.7, position=position_dodge())
+
  ggtitle("Average age of persons by state and sex")
```

Now the graph shows the average age in the bars:



There are lots of other possibilities within the *ggplot2* procedure, both when it comes to different chart types and layout. We can also use the plotly package for R. It is like *plotly* in Python, see the next chapter.

22.5. Python

In Python, we have several different modules for making graphs, for instance matplotlib, seaborn and plotly. We will now have a brief look at plotly (https://plotly.com/python/). With plotly, we can make quite nice graphs with just a little code.

We may have to install plotly before we can use it:

```
!pip install plotly
```

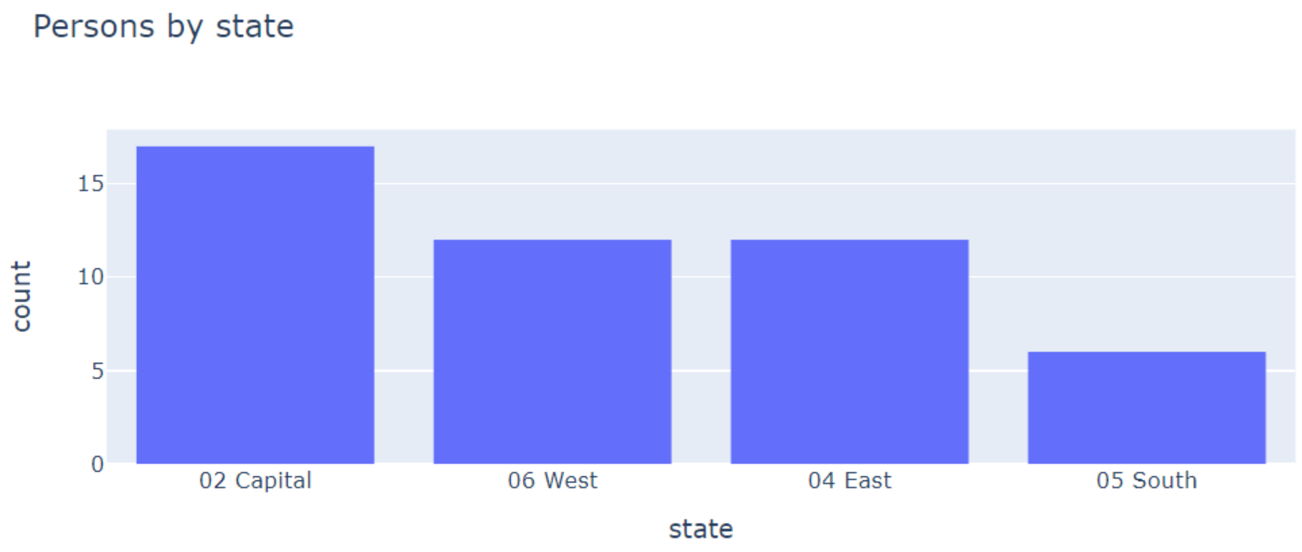
When plotly is installed, we import it:

```
import plotly.express as px
```

We start with a vertical bar chart for *state*, and will use *histogram* to make the graph:

```
px.histogram(mdgperson_nodup, x='state', title='Persons by state')
```

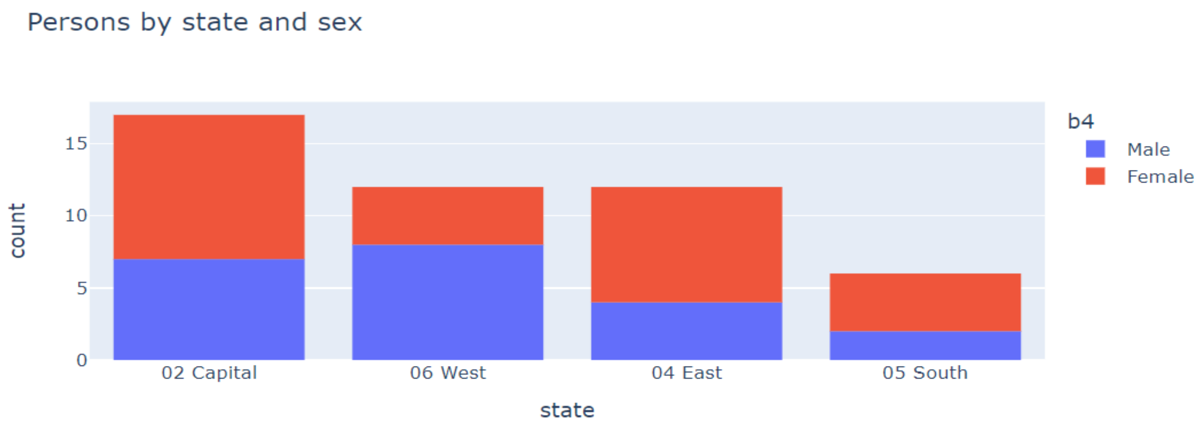
The graph looks like this:



We will add the column *b4* as a stacked variable in our chart:

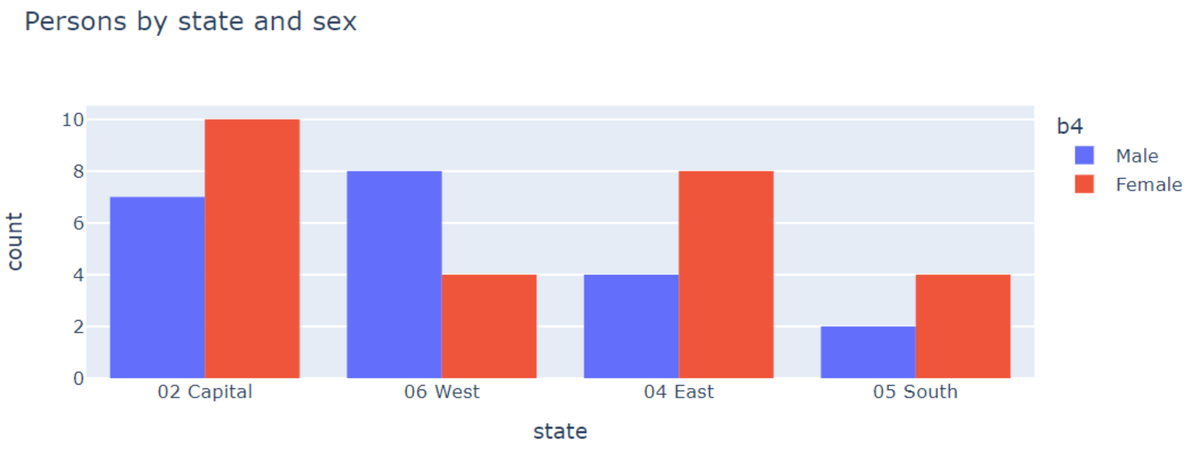
```
px.histogram(mdgperson_nodup, x='state', color='b4', title='Persons by state
and sex')
```

Now our graph looks like this:



We can put sex in separate bars in our chart by adding the *barmode* option:

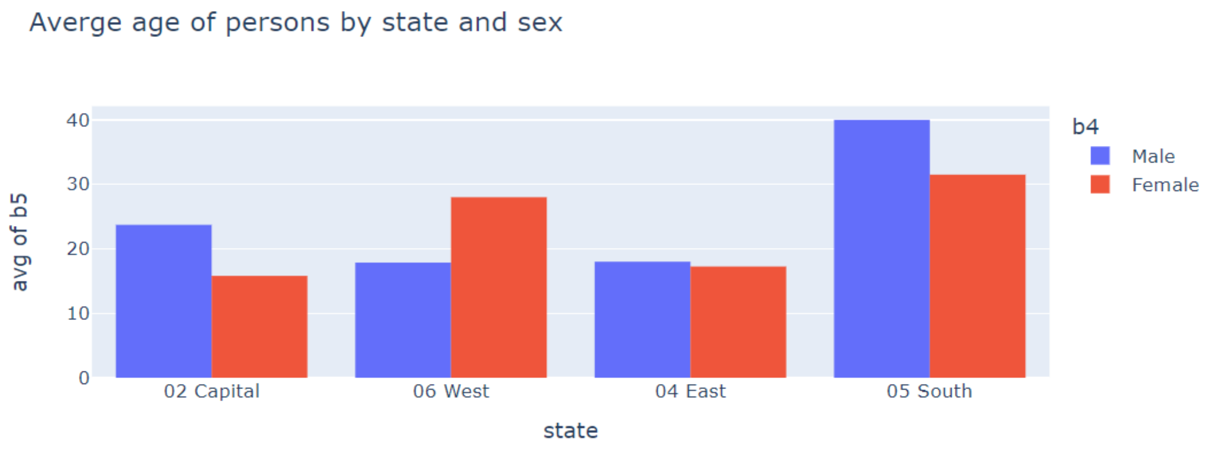
```
px.histogram(mdgperson_nodup, x='state', color='b4', barmode="group",
title='Persons by state and sex')
```



We may add a measure variable, like *b5,* and calculate the mean values:

```
px.histogram(mdgperson_nodup, x='state', y='b5', color='b4', histfunc='avg',
             barmode="group",
             title='Averge age of persons by state and sex')
```

Now we have average age in our graph:



There are lots of other possibilities for different chart types in plotly. We can also enhance the layout to a large extent.

23. Data exchange

As Sas, Spss, Stata, R and Python stores data in their own proprietary formats, we can't use datasets from one software in another without doing a data conversion. They can all import delimited or from fixed format files. Delimited files use a special character as a delimiter, like a comma, a semi-colon or a tabulator, between each column. However, we will now look at how the data exchange is made easier with built-in conversion of data files.

23.1. Sas

Sas has a possibility to read Spss and Stata datasets, but to do that we have to license the module "Access interface to PC file formats". Not all versions are compatible. When it comes to R, we can use the *Proc iml* procedure to import and export R data frames and also run R code.

Import

When the PC access module is licensed, we can import an Spss file like this:

```
proc import out=work.MDGPERSON_NODUP
  datafile = "h:\mdg\data\mdgperson_nodup.sav"
  dbms = SAV replace;
run;
```

The formats for the variables are also copied. However, they are stored in a temporary folder. They may be extracted from that temporary storage and put on a dataset like this:

```
proc format cntlout=formats;
run;
```

To save the formats with the other permanent formats we use this syntax (if a format exists it will overwrite it):

```
proc format cntlin=formats lib=library;
run;
```

When we have an Spss dataset with date and time variables and want to import it to Sas, we have to convert date variables and unformatted time variables. Even though we format date variables in Spss, Sas will import the Spss time values for that date. We can use a program like this to import the Spss dataset and convert the times and dates:

```
proc import out=times_from_spss
  file= "H:\MDG\Data\times.sav"
  dbms = SAV replace;
run;
data times_from_spss;
 set times_from_spss;
 date = datepart(date-11903760000);
 date2 = datepart(date2-11903760000);
 date_nf = datepart(date_nf - 11903760000);
 date2_nf = datepart(date2_nf - 11903760000);
 time_nf = time_nf - 11903760000;
 time2_nf = time2_nf - 11903760000;
 format date date2 yymmdd10. ;
run;
```

If we don't have a license for Access to PC file formats, the dataset should be exported to a Sas dataset in Spss before importing it to Sas, see page 205.

When it comes to importing Stata datasets, Sas version 9.4 can read Stata datasets at least up to release 16. If we cannot import the datasets, it might be because the Stata dataset may be stored in a higher version of Stata than Sas can read. If so, we must use the *saveold* command in Stata first:

```
use "mdgperson_nodup.dta", clear
saveold "mdgperson_nodup10.dta", replace
```

Then we import the dataset in the same way as Spss except for the *dbms* option:

```
proc import out=work.MDGPERSON_NODUP10
  datafile = "H:\MDG\Data\mdgperson_nodup10.dta"
  dbms = DTA replace;
run;
```

If the Stata dataset is saved in newer version than Sas can read, these messages are given in the log:

```
Didn't see end for |varnames| element.  Got -> ||

Requested Input File Is Invalid

ERROR: Import unsuccessful.  See SAS Log for details.
```

The formats can be copied the same way as for imported Spss datasets.

When we have a Stata dataset with time and date variables and import it to Sas, we have to convert all time variables and format them in Sas:

```
proc import out=times_from_stata
  file= "H:\MDG\Data\times.dta"
  dbms = DTA replace;
run;

data times_from_stata;
 set times_from_stata;
 time = time /1000;
 time2 = time2 /1000;
 time_nf = time_nf /1000;
 time2_nf = time2_nf /1000;
 format time time2 datetime19. ;
run;
```

There is no built-in functionality in Sas to import R data frames. However, we can call R from Sas. To be able to that, we have to add some settings. We must set the Rlang option in our sasv9.cfg file. The sasv9.cfg file is stored in a sub-folder where Sas is installed, usually something like this: c:\Program Files\SASHome\x86\SASFoundation\9.4\nls\en\sasv9.cfg. If we search our c-drive for sasv8.cfg, we might find more than one. Usually, we set the Rlang option in the stored in the nls\en subfolder. At the bottom of this file, we add this line:

```
-RLANG
```

When that is done and we have restarted Sas, we can add the path to R. Before we do that, we must know the path to the R installation we will use. We do that with *Sys.getenv* command in R:

```
Sys.getenv("R_HOME")
```

Then we can copy the path to our Sas program:

```
options set=R_HOME="C:/Users/krl/DOCUME~1/R/R-40~1.0";
```

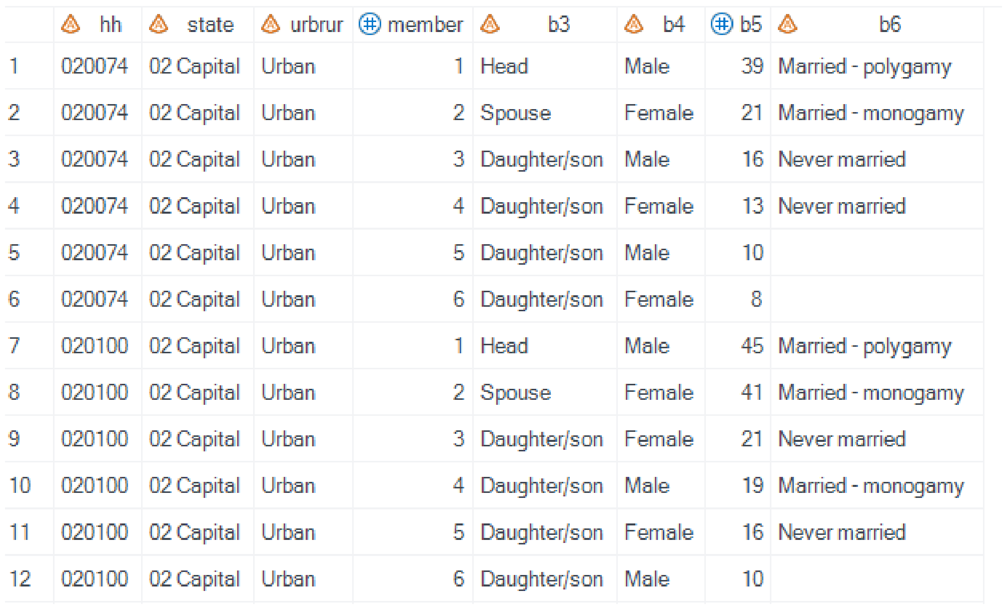
We can now use the *Proc iml* procedure in Sas and put our R code between the *submit/r* and *endsubmit* statements. Then we use the *call* statement to call the *ImportDataSetFromR* routine to convert from R data frame to Sas dataset:

```
proc iml;
submit/r;
load("H:/Mdg/Data/mdgperson_nodup.Rdata")
endsubmit;
call ImportDataSetFromR("mdgperson_nodup_sas","mdgperson_nodup");
quit;
```

To convert from the newer rds data format we use the *readRDS* command:

```
proc iml;
submit/r;
mdg <- readRDS("H:/Mdg/Data/mdgperson_nodup.rds")
endsubmit;
call ImportDataSetFromR("mdgperson_nodup_sas_rds","mdg");
quit;
```

The factor variables from the R data frame will we character variables in the Sas dataset where the levels from R will be values in Sas:



	hh	state	urbrur	member	b3	b4	b5	b6
1	020074	02 Capital	Urban	1	Head	Male	39	Married - polygamy
2	020074	02 Capital	Urban	2	Spouse	Female	21	Married - monogamy
3	020074	02 Capital	Urban	3	Daughter/son	Male	16	Never married
4	020074	02 Capital	Urban	4	Daughter/son	Female	13	Never married
5	020074	02 Capital	Urban	5	Daughter/son	Male	10	
6	020074	02 Capital	Urban	6	Daughter/son	Female	8	
7	020100	02 Capital	Urban	1	Head	Male	45	Married - polygamy
8	020100	02 Capital	Urban	2	Spouse	Female	41	Married - monogamy
9	020100	02 Capital	Urban	3	Daughter/son	Female	21	Never married
10	020100	02 Capital	Urban	4	Daughter/son	Male	19	Married - monogamy
11	020100	02 Capital	Urban	5	Daughter/son	Female	16	Never married
12	020100	02 Capital	Urban	6	Daughter/son	Male	10	

To keep the original values from the data frame it should be converted to a Sas dataset before the variables are changed to factors in R. However, the value formats then have to be added within Sas.

Sas version 9.4 cannot read Python data formats like pickle and parquet files. Instead, files should be converted to csv-files or similar in Python before importing to Sas. In Sas Viya though, it is possible. We can use the parquet engine in the libname statement and then read directly from a parquet file and save it as an Sas dataset:

```
libname pydata parquet 'h:\mdg\data\parquet';
data medgperson_nodup;
 set pydata.mdgperson_nodup;
run;
```

23.2. Export

It is also possible to export datasets from Sas to Spss or Stata formats. Then we must add a *fmtlib* statement if we want to bring the formats over. Here we export to Spss with formats:

```
proc export data=work.MDGPERSON_NODUP
  outfile= "H:\MDG\Data\mdgperson_nodup_from_sas.sav"
  dbms = SAV replace;
  fmtlib=library.formats;
run;
```

When we export date and time variables for use in Spss they should have formats connected before the export. For time variables we can use the datetime19. format and for date variables yymmdd10. is suitable:

```
format time time2 datetime19. date date2 yymmdd10.;
```

The syntax for export to Stata with formats is like this:

```
proc export data=mdg.MDGPERSON_NODUP
  outfile= "c:\temp\mdgperson_nodup_from_sas.dta"
  dbms = DTA replace;
  fmtlib=library.formats;
run;
```

However, we should avoid format names with more than 8 positions as they will not be added to the Stata datasets when we open them in Stata. If the numeric variables without value labels is less readable, we can change them like this:

```
use "mdgperson_nodup_from_sas.dta", clear
format member %1.0f
format b5 %2.0f
```

When we export time variables to an Spss dataset, formatted date and time variables will be converted correctly. If they are not formatted, we have to recalculate to get the values right, see page 204.

When we export to a Stata dataset with time variables, they will be converted to date variables if they have datetime formats in Sas. If we want to keep the whole time value, we should not use time formats in Sas. To drop formats in Sas we can use the *Format* statement in a Data step:

```
data times;
 set times;
 format time time2 ;
run;

proc export data=work.times
  outfile= "H:\MDG\Data\times_from_sas.dta"
  dbms = DTA replace;
run;
```

We must convert the time variables after we have imported the Stata dataset that was created by Sas, see page 209.

If we don't have the Access to PC file formats module licensed, we can export the dataset to a delimited file and then import that to Stata instead or we can use the Stata import sas functionality, see page 208.

To create an R file in Sas, we can use the R interface in Proc Iml. Before we do that, some settings in Sas must be added, see page 201.

Now we are ready to convert the Sas dataset to an R data frame. We use the Proc iml procedure in Sas and put our R code between the *submit/r* and *endsubmit* statements. First, we use the *call* statement to call the *ExportDataSetToR* routine to convert the Sas dataset to an R data frame. Then we use the *saveRDS* command to save the R data frame:

```
proc iml;
 call ExportDataSetToR("mdg.mdgperson_nodup","mdg_nodup_person_from_sas");
 submit/r;

saveRDS(mdg_nodup_person_from_sas,file="H:/Mdg/Data/mdgperson_nodup_from_sas.
rds")
 endsubmit;
quit;
```

Beware that formats are not included in the R data frame. If formats are needed it is better to import the Sas dataset in R, because then the formats are included as labels (see page 212).

When we want to import date and time variables from R, they are imported to Sas as dates and times if they are defined as date/time variables in R.

Sas version 9.4 cannot export to Python data formats like parquet or pickle. Instead, we can export to a csv-file or similar before we import the file in Python. In Sas Viya though, it is possible. We can use the parquet engine in the libname statement and then write directly to a parquet file:

```
libname pydata parquet 'h:/mdg/data/parquet';
data pydata.mdgperson_nodup;
 set mdg.mdgperson_nodup;
run;
```

23.3. Spss

Import

It is easy to import Sas datasets within Spss. We can use syntax like this:

```
GET SAS DATA='mdgperson_nodup.sas7bdat'.
```

However, value labels (formats) are not included with this syntax. Instead, we can add the formats with the subcommand *Formats*. Before we can do that, the formats have to be permanently stored in Sas (see page 67). To include the value labels, we do like this:

```
GET SAS DATA='mdgperson_nodup.sas7bdat' /FORMATS='..\cat\formats.sas7bcat'.
```

Sometimes, the formats will not be imported. It may be because of encoding problems. If so, it will be better to convert to Spss in Sas instead of trying to import a Sas dataset into Spss.

When we import date and time variables, the date variables are shown as we want. However, to show the time variables in a readable way we should add a *Formats* command in Spss:

```
FORMATS time time2 (DATETIME22).
```

Time variables which do not have a time format when they are exported from Sas will get wrong time values if we just add the *Formats* command in Spss. This is because Sas and Spss use different numbers to represent the same time. The same goes for date variables. These will also have to be adjusted. To adjust from Sas time and date variables to Spss time and date variables we have to re-compute the time variables:

```
GET SAS DATA='times.sas7bdat'.
COMPUTE time_nf =  11903760000 + time_nf.
COMPUTE time2_nf = 11903760000 + time2_nf.
COMPUTE date_nf = TIME.DAYS(date_nf+137775).
COMPUTE date2_nf = TIME.DAYS(date2_nf+137775).
EXECUTE.
FORMATS time time2 time_nf time2_nf (datetime22) date_nf date2_nf (adate10).
```

The *Time.days* function returns the day number since day 1 in the Spss day count.

To import a Stata dataset, we do it in a similar way:

```
GET STATA FILE='mdgperson_nodup.dta' .
```

Value labels in the Stata dataset are automatically included in the converted Spss dataset. Data files stored from Stata 16 are imported without problems. If the Stata version dataset is not supported, we get an error message like this:

```
Error # 7202.  Command name: GET STATA

Input dictionary read error.

Execution of this command stops.

Your new version of Stata is not supported
```

We must beware of the fact that all different missing values set in Sas and Stata will be converted to system missing in Spss.

When we import a Stata dataset with time and date variables to Spss we have to convert unformatted dates and all time variables. We can do it like this:

```
GET STATA FILE='times.dta'.
COMPUTE date_nf = TIME.DAYS(date_nf+137775).
COMPUTE date2_nf = TIME.DAYS(date2_nf+137775).
COMPUTE time =  11903760000 + (time/1000).
COMPUTE time2 = 11903760000 + (time2/1000).
COMPUTE time_nf =  11903760000 + (time_nf/1000).
COMPUTE time2_nf = 11903760000 + (time2_nf/1000).
EXECUTE.

FORMATS time time2 (datetime22).
FORMATS time_nf time2_nf date_nf date2_nf (f16).
SAVE OUTFILE='times_from_stata.sav'.
```

There is now built-in functionality to import an R or Python data frame to an Spss dataset. It is easy to read and write Spss datasets in R or Python, so it should be done there instead (see page 213).

Export

To export Spss data to Sas we use the *Save translate* command. When it comes to value labels, we can add them with the subcommand *Valfile*.

```
GET FILE='mdgperson_nodup.sav'.
SAVE TRANSLATE OUTFILE='mdgperson_nodup_from_spss.sas7bdat'
```

```
  /TYPE=SAS
  /VERSION=7
  /PLATFORM=WINDOWS
  /MAP
  /REPLACE
  /VALFILE='..\syntax\formats.sas'.
```

Here we name a syntax file that will include the Sas syntax needed to create the formats and the connection to the data. This syntax file will assume that the formats and the datasets are in the same folder which they should not. We have to change a little bit in the Sas syntax before the program is executed, we end up with this:

```
proc format library = library ;
   value STATE
      1 = '01 Central'
      2 = '02 Capital'
      3 = '03 North'
      4 = '04 East'
      5 = '05 South'
      6 = '06 West' ;
   value URBRUR
      1 = 'Urban'
      2 = 'Rural' ;
   value B3F
      0 = 'Head'
      1 = 'Spouse'
      2 = 'Daughter/son'
      3 = 'Spouse of son/daughter'
      4 = 'Grandchild'
      5 = 'Sister/brother'
      6 = 'Sister/brother in-laws'
      7 = 'Parent'
      8 = 'Parent-in-law'
      9 = 'Niece/nephew'
      10 = 'Other relative'
      11 = 'Non relative' ;
   value B4F
      1 = 'Male'
      2 = 'Female' ;
   value B6F
      1 = 'Never married'
      2 = 'Married - monogamy'
      3 = 'Married - polygamy'
      4 = 'Widowed'
      5 = 'Separated'
      6 = 'Divorced' ;
run;

proc datasets library = mdg nolist;
modify mdgperson_nodup_from_spss;
   format    state STATE.;
   format    urbrur URBRUR.;
   format    b3 B3F.;
   format    b4 B4F.;
   format    b6 B6F.;
quit;
```

The definitions of the *librefs* Library and Mdg are assigned earlier in the Autoexec flow.

If there are problems with adding the formats, it might be encoding mismatches. An error message like this will be shown in the log:

```
ERROR: File MDG.MDGPERSON_NODUP_FROM_SPSS cannot be updated because its
encoding does not match the session encoding or the file is

       in a format native to another host, such as WINDOWS_32.
```

We can add the formats this way instead:

```
data mdg.mdgperson_nodup_from_spss;
 set mdg.mdgperson_nodup_from_spss;
 format state STATE.;
 format urbrur URBRUR.;
 format b3 B3F.;
 format b4 B4F.;
 format b6 B6F.;
run;
```

To export Spss dataset to Stata we also use the *Save translate* command. Here the value labels are automatically added to the Stata dataset. We use this syntax:

```
SAVE TRANSLATE OUTFILE='mdgperson_nodup_from_spss.dta'
  /TYPE=STATA
  /MAP
  /REPLACE.
```

Both system missing and user missing values are converted to the missing value dot (.) in converted Sas and Stata datasets.

When we export datasets with time and date variables, we should not use formats for the time variables as Stata will convert them to date variables. We change the formats before the export:

```
GET FILE='times.sav'.
FORMATS time time2 time_nf time2_nf (f16).

SAVE TRANSLATE OUTFILE='times_from_spss.dta'
  /TYPE=STATA
  /MAP
  /REPLACE.
```

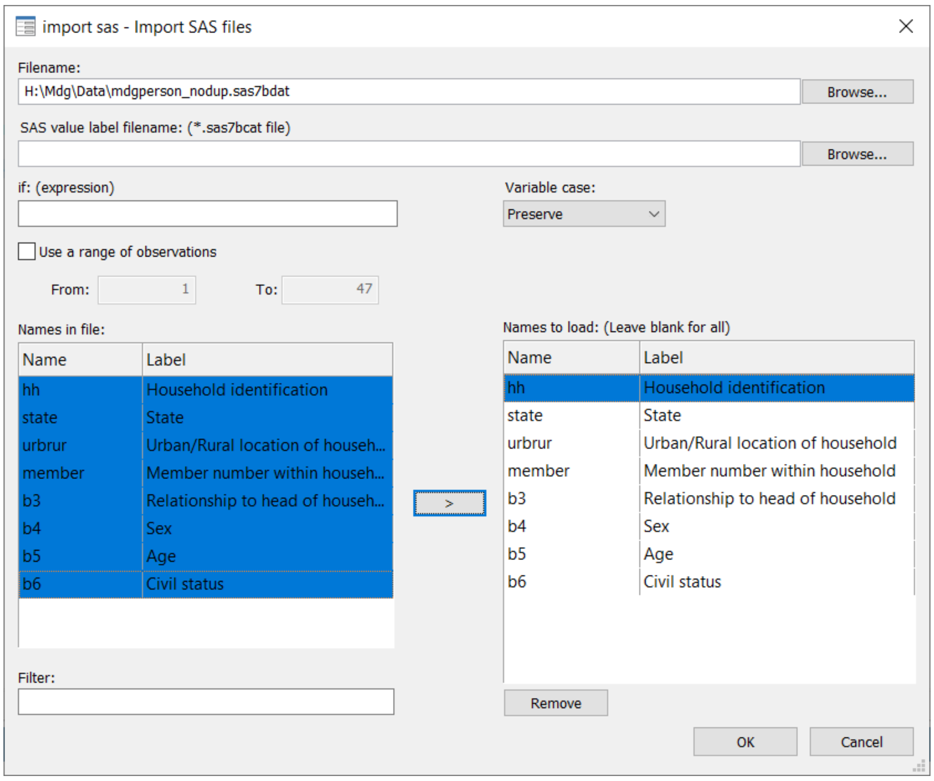
The time variables will have to be converted after they are converted to a Stata dataset, see page 209.

There is now built-in functionality to export an Spss dataset to an R or Python data frame. It is easy to read and write Spss datasets in R and Python, so it should be done there instead (see page 213).

23.4. Stata

Import

Before version 16 of Stata there was no easy way to store Stata datasets as Sas or Spss datasets within Stata. Beginning with version 16, import functions are added. We can use the interface from the File➔Import menu and choose Sas or Spss data. A window will appear when we have chosen our import file. We can mark all our variables (CTRL+A) and click on the arrow to include them in the import:



import sas - Import SAS files

Filename: H:\Mdg\Data\mdgperson_nodup.sas7bdat

SAS value label filename: (*.sas7bcat file)

if: (expression)

Variable case: Preserve

Use a range of observations

From: 1 To: 47

Names in file:

Name	Label
hh	Household identification
state	State
urbrur	Urban/Rural location of househ...
member	Member number within househ...
b3	Relationship to head of househ...
b4	Sex
b5	Age
b6	Civil status

Names to load: (Leave blank for all)

Name	Label
hh	Household identification
state	State
urbrur	Urban/Rural location of household
member	Member number within household
b3	Relationship to head of household
b4	Sex
b5	Age
b6	Civil status

Filter:

When we click OK, the import will be executed. The command will appear in the Results window. This may be copied into a do file if we like. It looks like this:

```
import sas hh state urbrur member b3 b4 b5 b6 using
"H:\Mdg\Data\mdgperson_nodup.sas7bdat", clear
```

However, when we use syntax, we don't need to mention the variable names when we want to import them all:

```
import sas using "H:\Mdg\Data\mdgperson_nodup.sas7bdat", clear
```

To import formats, we can add the name of the format catalog name with the *bcat* option. To do this we have to use syntax. After the import we must add the value labels to the variables. To find the value labels names we can run the *label list* command. Then we can add them with the *label values* command:

```
import sas using "H:\Mdg\Data\mdgperson_nodup.sas7bdat",
bcat("H:\Mdg\Cat\formats.sas7bcat") clear
label list

label values state STATE
label values urbrur URBRUR
label values b3 HEAD_REL
label values b4 SEX
label values b6 CIVIL_STATUS
```

As only numeric variables can have value labels in Stata, formats for character variables cause problems. If there are character formats in the formats.sas7bcat file, we can get an note like this:

```
note: invalid numeric value for value label CIVIL_STATUS, skipped
```

Even though it is only a note, the import will not be executed. The note says it is a problem with one of the numeric formats, however the problem is that there is at least one character format in the formats.sas7bcat file. This must be deleted from the formats.sas7bcat file before we can use it in Stata. Here is an example on how we delete the character format $sex in Sas:

```
proc catalog catalog=library.formats;
  delete sex.formatc;
quit;
```

Missing values and the *Other* group in Sas formats will not be included in the value labels in Stata.

We might get an error message if some data cannot be read because of different encoding. This was thrown when there were problems with a special dash in the formats:

```
may not label -2.0603e-289
```

To solve this problem, we must change the value in Sas before the formats are imported to Stata.

Stata counts time in milliseconds and Sas in seconds with milliseconds as decimals. This means we have to multiply unformatted date values with 1000 to get the right times in Stata. Stata and Sas use the same numbers for the dates, so it does not matter if these date variables have no formats. In Stata we should do like this to correct the date and time values:

```
use "times_from_sas.dta", clear
replace time = time * 1000
replace time2 = time2 * 1000
replace time_nf = time_nf * 1000
replace time2_nf = time2_nf * 1000
format time %tc
format time2 %tc
format time_nf %tc
format time2_nf %tc
format date_nf %d
format date2_nf %d
```

The time variables which were formatted in Sas will still just contain the date when imported to Stata.

The process is the same for Spss datasets. The converted datasets will include value labels:

```
import spss using "H:\Mdg\Data\mdgperson_nodup.sav", clear
```

When importing Stata datasets made in Spss with unformatted time variables we will convert them like this in Stata:

```
use "times_from_spss.dta", clear
replace time = (time - 11903760000)*1000
replace time_nf = (time_nf - 11903760000)*1000
replace time2 = (time2 - 11903760000)*1000
replace time2_nf = (time2_nf - 11903760000)*1000
format time time2 %tc
format time_nf time2_nf %17.0f
replace date_nf = dofc((date_nf - 11903760000)*1000)
replace date2_nf = dofc((date2_nf - 11903760000)*1000)
```

There are no import built-in functions in Stata to import files from R or Python. However, in R we can convert a data frame to a Stata dataset, and we can also convert a Stata dataset to an R data frame, see page 215. In Python, we can export to Stata with the to_stata function, see page 220.

23.5. Export

It is possible to export from Stata to Sas, but only as an export file. The export file may be in version 5 or 8 of Sas. With version 5, formats are also exported (as long as the value labels names in Stata are 8 or less characters).

Here is an example for export to Sas version 5 xport file:

```
export sasxport5 "h:\mdg\data\mdgperson_nodup_from_stata5.v5xpt", replace
```

With this program there will be created a data file and a file with the Sas formats. We must convert them in Sas to ordinary Sas datasets and formats and create Sas formats from the format xpt file. We can use syntax like this to convert:

```
libname xptform xport 'h:\mdg\data\formats.xpf';
libname xptfile xport 'h:\mdg\data\mdgperson_nodup_from_stata5.v5xpt';

proc copy in = xptform out = work ;
run;

proc format cntlin=work.formats;
run;

proc copy in = xptfile out = work ;
run;
```

We can also use the *sasxport8* option for our export. If we want to export the formats as well, we add the *vallabfile* option:

```
export sasxport8 "h:\mdg\data\mdgperson_nodup_from_stata8.v8xpt", vallabfile replace
```

In addition to the exported data file, Stata will create a Sas program with syntax for the import of the data and creation of Sas formats. This file will be stored in the same folder as the data file. However, the import program does not work for xport files higher than version 5 of Sas. Hence, we must convert it in another way. Sas has made a built-in macro for this which we will use. This macro will create a Sas dataset named *dataset*, with no formats attached. We will create the formats from the Sas program. We can add the formats and save the dataset with a new name:

```
%xpt2loc(libref=work,
         filespec='h:\mdg\data\mdgperson_nodup_from_stata8.v8xpt' );
proc format library = work ;
      value STATE
            1 = '01 Central'
            2 = '02 Capital'
            3 = '03 North'
            4 = '04 East'
            5 = '05 South'
            6 = '06 West' ;
      value URBRUR
            1 = 'Urban'
            2 = 'Rural' ;
      value HEAD_REL
            0 = 'Head'
            1 = 'Spouse'
            2 = 'Daughter/son'
            3 = 'Spouse of son/daughter'
            4 = 'Grandchild'
            5 = 'Sister/brother'
            6 = 'Sister/brother in-laws'
            7 = 'Parent'
            8 = 'Parent-in-law'
            9 = 'Niece/nephew'
            10 = 'Other relative'
            11 = 'Non relative' ;
      value SEX
            1 = 'Male'
            2 = 'Female' ;
      value CIVIL_STATUS
            1 = 'Never married'
            2 = 'Married - monogamy'
            3 = 'Married - polygamy'
            4 = 'Widowed'
            5 = 'Separated'
            6 = 'Divorced' ;
run ;
data mdgperson_nodup_from_stata;
 set dataset;
 format
  state  state.
  urbrur urbrur.
  b3     head_rel.
  b4     sex.
  b6     civil_status.
  ;
run;
```

Date variables are converted correctly, but not time and datetime variables. We must divide them by 1000 to get the correct values. Date formats from the Stata dataset will be added to date variables, but not datetime and time variables. We can import, convert, and correct datetime and time variables like this:

```
libname xptfile xport 'h:\mdg\data\times_from_stata5.v5xpt';
proc copy in = xptfile out = work ;
run;
data times_from_stata5;
 set times_fr;
 time = time /1000;
 time2 = time2 /1000;
 time_nf = time_nf /1000;
 time2_nf = time2_nf /1000;
 format time time2 datetime19. ;
run;
```

Stata does not have any built-in functionality for exporting a Stata dataset to an Spss dataset. Instead, we can import a Stata dataset to Spss, see page 224.

Stata does not have any built-in procedures to export files to R or Python. Instead, we can read Stata files in R, by using the Haven package to import Stata datasets, see page 214. In Python, we can use pd.read_stata to import Stata files.

23.6. R

One common R package to read and write data files from other systems is *haven* (which is also included in the *tidyverse* package). After installation and attaching the package, we can import and export files. First, we have to attach the package:

```
library(haven)
```

Import

After attaching the *haven* package, we can import data from other systems. We start with importing a Sas dataset:

```
mdgfromsas <- read_sas(data_file="h:/mdg/data/mdgperson_nodup.sas7bdat")
```

When Sas formats are connected to the Sas dataset we can add them, they are stored in a file called formats.sas7bcat:

```
mdgfromsaswithformats <- read_sas(data_file="h:/mdg/data/mdgperson_nodup.sas7
bdat",catalog_file ="h:/mdg/cat/formats.sas7bcat")
```

The formats are now labels for each variable:

```
mdgfromsaswithformats
```

```
# A tibble: 47 x 8

   hh               state    urbrur member              b3         b4    b5                    b6

   <chr>        <dbl+lbl> <dbl+lbl>  <dbl>        <dbl+lbl>  <dbl+lbl> <dbl>              <dbl+lbl>

 1 020074  2 [02 Capital] 1 [Urban]      1 0 [Head]          1 [Male]      39  3 [Married - polygamy]
 2 020074  2 [02 Capital] 1 [Urban]      2 1 [Spouse]        2 [Female]    21  2 [Married - monogamy]
 3 020074  2 [02 Capital] 1 [Urban]      3 2 [Daughter/son]  1 [Male]      16  1 [Never married]
 4 020074  2 [02 Capital] 1 [Urban]      4 2 [Daughter/son]  2 [Female]    13  1 [Never married]
 5 020074  2 [02 Capital] 1 [Urban]      5 2 [Daughter/son]  1 [Male]      10 NA
 6 020074  2 [02 Capital] 1 [Urban]      6 2 [Daughter/son]  2 [Female]     8 NA
 7 020100  2 [02 Capital] 1 [Urban]      1 0 [Head]          1 [Male]      45  3 [Married - polygamy]
 8 020100  2 [02 Capital] 1 [Urban]      2 1 [Spouse]        2 [Female]    41  2 [Married - monogamy]
 9 020100  2 [02 Capital] 1 [Urban]      3 2 [Daughter/son]  2 [Female]    21  1 [Never married]
10 020100  2 [02 Capital] 1 [Urban]      4 2 [Daughter/son]  1 [Male]      19  2 [Married - monogamy]

# ... with 37 more rows
```

However, these labels are a little inconvenient to use. Hence, we can convert the variables to factors with levels and labels. We use the as_factor function since the as.factor function does not work here. The labels are attributes to each variable and the attribute is called labels:

```
mdgfromsaswithformats$state <- as_factor(mdgfromsaswithformats$state,levels="
labels")
mdgfromsaswithformats$urbrur <- as_factor(mdgfromsaswithformats$urbrur,levels
="labels")
mdgfromsaswithformats$b3 <- as_factor(mdgfromsaswithformats$b3,levels="labels
")
mdgfromsaswithformats$b4 <- as_factor(mdgfromsaswithformats$b4,levels="labels
")
mdgfromsaswithformats$b6 <- as_factor(mdgfromsaswithformats$b6,levels="labels
")
```

Now the data frame is like this:

```
mdgfromsaswithformats
```

```
# A tibble: 47 x 8

   hh     state      urbrur member b3           b4         b5 b6

   <chr>  <fct>      <fct>   <dbl> <fct>        <fct>   <dbl> <fct>

 1 020074 02 Capital Urban       1 Head         Male       39 Married - polygamy
 2 020074 02 Capital Urban       2 Spouse       Female     21 Married - monogamy
 3 020074 02 Capital Urban       3 Daughter/son Male       16 Never married
 4 020074 02 Capital Urban       4 Daughter/son Female     13 Never married
 5 020074 02 Capital Urban       5 Daughter/son Male       10 NA
 6 020074 02 Capital Urban       6 Daughter/son Female      8 NA
 7 020100 02 Capital Urban       1 Head         Male       45 Married - polygamy
 8 020100 02 Capital Urban       2 Spouse       Female     41 Married - monogamy
 9 020100 02 Capital Urban       3 Daughter/son Female     21 Never married
10 020100 02 Capital Urban       4 Daughter/son Male       19 Married - monogamy

# ... with 37 more rows
```

The values behind the levels will change if the original values did not start from 1 and increment with 1 (as discussed earlier, page 62.

When it comes to date and time variables, they will be converted if they are formatted as date or time in Sas. If not, the number behind the date or time will be imported:

```
times <- read_sas(data_file="h:/mdg/data/times.sas7bdat")
```

We can look at the data frame in R:

```
times
```

```
# A tibble: 12 x 9

      id time                time2               date       date2         time_nf   time2_nf date_nf date2_nf

   <dbl> <dttm>              <dttm>              <date>     <date>          <dbl>      <dbl>   <dbl>    <dbl>

1      1 1999-03-01 11:42:00 1962-04-24 18:25:31 1962-12-15 2005-04-16 1235907720   72987931    1079    16542
2      2 2002-12-25 02:40:12 1954-03-09 15:35:26 1961-09-03 1990-09-24 1356403212 -183457474     611    11224
3      3 1973-08-02 03:27:41 1962-11-30 08:56:23 1962-04-25 1966-10-01  428729261   91961783     845     2465
4      4 1984-04-08 17:06:49 1935-04-04 15:34:40 1962-11-16 1979-04-14  765911209 -780827120    1050     7043
5      5 2003-02-04 19:42:52 1963-02-18 06:53:54 1962-12-04 2013-03-17 1360006972   98866434    1068    19434
6      6 1966-09-02 09:37:17 1935-11-02 16:23:38 1962-01-10 2000-07-13  210505037 -762507382     740    14804
7      7 1969-09-26 22:23:10 1964-09-19 22:15:04 1960-06-28 1986-09-18  307318990  148947304     179     9757
8      8 1995-02-10 20:17:57 1963-02-08 06:09:53 1964-03-01 1984-05-10 1108066677   97999793    1521     8896
9      9 1970-09-01 17:19:39 1970-09-01 13:58:32 1960-12-20 2018-01-05  336676779  336664712     354    21189
10    10 1979-12-20 04:57:52 1963-06-22 05:37:12 1963-01-21 1985-04-07  630133072  109575432    1116     9228
11    11 2002-07-19 18:23:49 NA                  1960-10-02 1960-08-21 1342722229         NA     275      233
12    12 2001-09-04 03:25:32 1961-09-28 03:25:32 1963-09-19 1963-09-19 1315193132   54962732    1357     1357
```

We can convert date variables that are not formatted in Sas with the *format* and *as.Date* functions. We use the *origin* argument to tell which date is date 0 in Sas (1 December 1960). For datetime variables that are not formatted in Sas, we can convert using the *as.POSIXct* function. The default time zone is taken from *Sys.timezone()*, but may be set with the *tz* argument, for instance tz="GMT". Here is an example:

```
times$time_nf <- as.POSIXct(times$time_nf, origin = "1960-01-01")
times$time_nf2 <- as.POSIXct(times$time2_nf, origin = "1960-01-01",tz=Sys.tim
ezone())
times$date_nf <- format(as.Date(times$date_nf, origin="1960-01-01"),"%Y-%m-%d
")
times$date2_nf <- format(as.Date(times$date2_nf, origin="1960-01-01"),"%Y-%m-
%d")
```

To import Spss files is similar to Sas datasets. We use the *read_spss* command. However, as value labels are stored in the Spss file they are included in the import:

```
mdgfromspss <- read_spss(file="h:/mdg/data/mdgperson_nodup.sav")
```

The data frame will be like the one read from Sas. That means we can convert the labelled variables to factors:

```
mdgfromspss$state <- as_factor(mdgfromspss$state,levels="labels")
mdgfromspss$urbrur <- as_factor(mdgfromspss$urbrur,levels="labels")
mdgfromspss$b3 <- as_factor(mdgfromspss$b3,levels="labels")
mdgfromspss$b4 <- as_factor(mdgfromspss$b4,levels="labels")
mdgfromspss$b6 <- as_factor(mdgfromspss$b6,levels="labels")
```

When we convert an Spss dataset with dates and times, we have to convert the Spss time number to R date and/or time. Then we have to set the origin to October 14, 1582 as that is date zero in Spss. All time and dates are counted in seconds from that time, so we must divide with 86400 if we want to convert to a date. There may also be some issues with time zones and daylight-saving time. This can lead to some different hours than the original. Here is an example:

```
times_spss <- read_sav(file="h:/mdg/data/times.sav")
times_spss$time_nf <- as.POSIXct(times_spss$time_nf, origin = "1582-10-14")
times_spss$time2_nf <- as.POSIXct(times_spss$time2_nf, origin = "1582-10-14")
times_spss$date_nf <- format(as.Date(times_spss$date_nf/86400, origin="1582-1
0-14"),"%Y-%m-%d")
times_spss$date2_nf <- format(as.Date(times_spss$date2_nf/86400, origin="1582
-10-14"),"%Y-%m-%d")
times_spss
```

The output dataset:

```
# A tibble: 12 x 9

      id time                time2               date       date2      time_nf             time2_nf            date_nf    date2_nf

   <dbl> <dttm>              <dttm>              <date>     <date>     <dttm>              <dttm>              <chr>      <chr>

 1     1 1999-03-01 11:42:00 1962-04-24 18:25:31 1962-12-15 2005-04-16 1999-03-01 12:42:00 1962-04-24 19:25:31 1962-12-15 2005-04-16
 2     2 2002-12-25 02:40:12 1954-03-09 15:35:26 1961-09-03 1990-09-24 2002-12-25 03:40:12 1954-03-09 16:35:26 1961-09-03 1990-09-24
 3     3 1973-08-02 03:27:41 1962-11-30 08:56:23 1962-04-25 1966-10-01 1973-08-02 04:27:41 1962-11-30 09:56:23 1962-04-25 1966-10-01
 4     4 1984-04-08 17:06:49 1935-04-04 15:34:40 1962-11-16 1979-04-14 1984-04-08 19:06:49 1935-04-04 16:34:40 1962-11-16 1979-04-14
 5     5 2003-02-04 19:42:52 1963-02-18 06:53:54 1962-12-04 2013-03-17 2003-02-04 20:42:52 1963-02-18 07:53:54 1962-12-04 2013-03-17
 6     6 1966-09-02 09:37:17 1935-11-02 16:23:38 1962-01-10 2000-07-13 1966-09-02 10:37:17 1935-11-02 17:23:38 1962-01-10 2000-07-13
 7     7 1969-09-26 22:23:10 1964-09-19 22:15:04 1960-06-28 1986-09-18 1969-09-26 23:23:10 1964-09-19 23:15:04 1960-06-28 1986-09-18
 8     8 1995-02-10 20:17:57 1963-02-08 06:09:53 1964-03-01 1984-05-10 1995-02-10 21:17:57 1963-02-08 07:09:53 1964-03-01 1984-05-10
 9     9 1970-09-01 17:19:39 1970-09-01 13:58:32 1960-12-20 2018-01-05 1970-09-01 18:19:39 1970-09-01 14:58:32 1960-12-20 2018-01-05
10    10 1979-12-20 04:57:52 1963-06-22 05:37:12 1963-01-21 1985-04-07 1979-12-20 05:57:52 1963-06-22 06:37:12 1963-01-21 1985-04-07
11    11 2002-07-19 18:23:49 NA                  1960-10-02 1960-08-21 2002-07-19 20:23:49 NA                  1960-10-02 1960-08-21
12    12 2001-09-04 03:25:32 1961-09-28 03:25:32 1963-09-19 1963-09-19 2001-09-04 05:25:32 1961-09-28 04:25:32 1963-09-19 1963-09-19
```

We see that time and time_nf differs with one to two hours.

We use *read_dta* to import a Stata file. We can also convert labelled variables to factors:

```
mdgfromstata <- read_dta(file="h:/mdg/data/mdgperson_nodup.dta")
mdgfromstata$state <- as_factor(mdgfromstata$state,levels="labels")
mdgfromstata$urbrur <- as_factor(mdgfromstata$urbrur,levels="labels")
mdgfromstata$b3 <- as_factor(mdgfromstata$b3,levels="labels")
mdgfromstata$b4 <- as_factor(mdgfromstata$b4,levels="labels")
mdgfromstata$b6 <- as_factor(mdgfromstata$b6,levels="labels")
```

For Stata, it is similar to Spss when it comes to datetime variables. However, the time is counted in milliseconds, not seconds with decimals as Spss does. This means we have to divide with 1000 to convert the datetime values. Still there may be some time zone problems here as well. For date variables, we must set the original date 0 to January 1, 1960:

```
times_stata <- read_dta(file="h:/mdg/data/times.dta")
times_stata$time_nf <- as.POSIXct(times_stata$time_nf/1000, origin = "1960-01
-01")
times_stata$time2_nf <- as.POSIXct(times_stata$time2_nf/1000, origin = "1960-
01-01")
times_stata$date_nf <- format(as.Date(times_stata$date_nf, origin="1960-01-01
"),"%Y-%m-%d")
times_stata$date2_nf <- format(as.Date(times_stata$date2_nf, origin="1960-01-
01"),"%Y-%m-%d")
```

```
times_stata
```

Here, time and time_nf differs with up to two hours:

```
      id time                time2               date       date2      time_nf             time2_nf            date_nf    date2_nf

   <dbl> <dttm>              <dttm>              <date>     <date>     <dttm>              <dttm>              <chr>      <chr>

 1     1 1999-03-01 11:42:00 1962-04-24 18:25:31 1962-12-15 2005-04-16 1999-03-01 12:42:00 1962-04-24 19:25:31 1962-12-15 2005-04-16

 2     2 2002-12-25 02:40:12 1954-03-09 15:35:26 1961-09-03 1990-09-24 2002-12-25 03:40:12 1954-03-09 16:35:26 1961-09-03 1990-09-24
 3     3 1973-08-02 03:27:41 1962-11-30 08:56:23 1962-04-25 1966-10-01 1973-08-02 04:27:41 1962-11-30 09:56:23 1962-04-25 1966-10-01
 4     4 1984-04-08 17:06:49 1935-04-04 15:34:40 1962-11-16 1979-04-14 1984-04-08 19:06:49 1935-04-04 16:34:40 1962-11-16 1979-04-14
 5     5 2003-02-04 19:42:52 1963-02-18 06:53:54 1962-12-04 2013-03-17 2003-02-04 20:42:52 1963-02-18 07:53:54 1962-12-04 2013-03-17
 6     6 1966-09-02 09:37:17 1935-11-02 16:23:38 1962-01-10 2000-07-13 1966-09-02 10:37:17 1935-11-02 17:23:38 1962-01-10 2000-07-13
 7     7 1969-09-26 22:23:10 1964-09-19 22:15:04 1960-06-28 1986-09-18 1969-09-26 23:23:10 1964-09-19 23:15:04 1960-06-28 1986-09-18
 8     8 1995-02-10 20:17:57 1963-02-08 06:09:53 1964-03-01 1984-05-10 1995-02-10 21:17:57 1963-02-08 07:09:53 1964-03-01 1984-05-10
 9     9 1970-09-01 17:19:39 1970-09-01 13:58:32 1960-12-20 2018-01-05 1970-09-01 18:19:39 1970-09-01 14:58:32 1960-12-20 2018-01-05
10    10 1979-12-20 04:57:52 1963-06-22 05:37:12 1963-01-21 1985-04-07 1979-12-20 05:57:52 1963-06-22 06:37:12 1963-01-21 1985-04-07
11    11 2002-07-19 18:23:49 NA                  1960-10-02 1960-08-21 2002-07-19 20:23:49 NA                  1960-10-02 1960-08-21
12    12 2001-09-04 03:25:32 1961-09-28 03:25:32 1963-09-19 1963-09-19 2001-09-04 05:25:32 1961-09-28 04:25:32 1963-09-19 1963-09-19
```

We can import parquet files from Python with the read_parquet command in the arrow library. This library is not included in the basic R installation, so we have to install it first. With internet connection we can install it like this:

```
install.packages('arrow')
```

When it is properly installed, we can activate it:

```
library(arrow)
```

Now we can import a parquet file to R:

```
mdgperson_from_parquet <- read_parquet('h:/mdg/data/mdgperson.parquet')
```

Export

To export a R data frame to a Sas dataset we could use the *write_sas* command. However, it is experimental and doesn't seem to work. Instead, we can run R from Sas and convert from R to Sas within *Proc iml*, see page 202.

To write to an Spss dataset we can use the write_sav command.

```
write_sav(data=mdgperson_nodup,path="h:/mdg/data/mdgperson_nodup_from_r.sav",
compress = TRUE)
```

This will create a compressed Spss dataset which can be opened with this command in SPSS. Variable and value labels will be included if they are stored in the R data frame. Factor variables with levels will be converted to numeric variables with value labels. The same goes for numeric variables with label attributes.

If we don't want the Spss dataset to be compressed, we can omit the *compress* argument or set it to FALSE.

To open the dataset in Spss we can use the *GET* command:

```
GET FILE ="h:\mdg\data\mdgperson_nodup_from_r.sav" .
```

We can use the *write_dta* command to convert a R data frame to a Stata dataset:

```
write_dta(data=mdgperson_nodup,path="h:/mdg/data/mdgperson_nodup_from_r.dta")
```

It converts factors variables with levels and numeric variables with label attributes to numeric variables with value labels.

To open the converted Stata dataset, we can use a command like this:

```
use "h:\mdg\data\mdgperson_nodup_from_r.dta", clear
```

We can write parquet files from R. To do that, we need the arrow package. It is not included in the basic R installation, so page 215 how to install it. We activate the library and do our export:

```
library(arrow)
write_parquet(mdgperson_nodup, 'h:/mdg/data/mdgperson_from_r.parquet')
```

23.7. Python

Import

We can use Pandas read_sas to read a Sas dataset into a Python data frame. But that demands that pyreadstat is installed and imported:

```
!pip install pyreadstat
import pyreadstat
```

When it is installed, it is easy to import:

```
mdgperson_from_sas = pd.read_sas(datapath + 'mdgperson.sas7bdat')
```

For variables with value labels, these are not converted into category variables. Only the codes are imported. Character variables are sometimes, if the locale of the Sas dataset is different from what is used in Python, surrounded by b''. To avoid these annoyances, we can use this syntax instead where we add the encoding parameter:

```
mdgperson_from_sas = pd.read_sas(datapath + 'mdgperson.sas7bdat',
 encoding ='iso-8859-1')
```

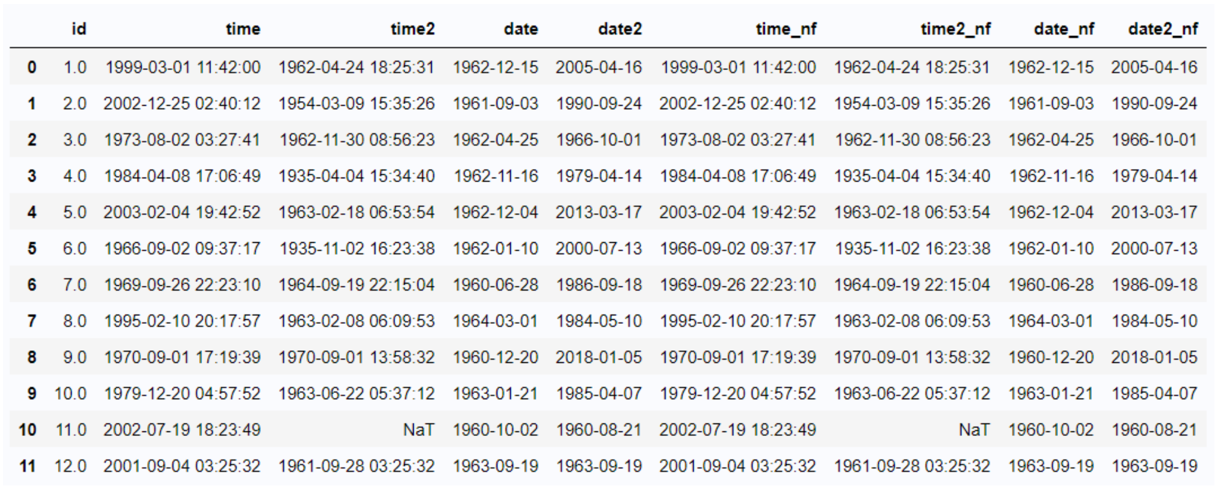
Another way to do this is to use pyreadstat.read_sas7bdat instead:

```
mdgperson_from_sas, meta = pyreadstat.read_sas7bdat(datapath +
'mdgperson.sas7bdat')
```

Date and time variables will be converted correctly as long as they are formatted in Sas with a datetime or date format. Unformatted date and time variables will have the number behind the date and time imported. We can convert these numbers to datetime variables in Python. One way to do it is to use the Pandas *timedelta* function and count the seconds from the time 0 in Sas which is January 1, 1960.

```
times_sas, meta = pyreadstat.read_sas7bdat(datapath + 'times.sas7bdat')
times_sas['time_nf'] = pd.to_timedelta(times_sas['time_nf'], unit='s') +
pd.Timestamp('1960-01-01')
times_sas['time2_nf'] = pd.to_timedelta(times_sas['time2_nf'], unit='s') +
pd.Timestamp('1960-01-01')
times_sas['date_nf'] = pd.to_timedelta(times_sas['date_nf'], unit='D') +
pd.Timestamp('1960-01-01')
times_sas['date2_nf'] = pd.to_timedelta(times_sas['date2_nf'], unit='D') +
pd.Timestamp('1960-01-01')
times_sas
```

The imported and converted data frame with converted datetime variables:



	id	time	time2	date	date2	time_nf	time2_nf	date_nf	date2_nf
0	1.0	1999-03-01 11:42:00	1962-04-24 18:25:31	1962-12-15	2005-04-16	1999-03-01 11:42:00	1962-04-24 18:25:31	1962-12-15	2005-04-16
1	2.0	2002-12-25 02:40:12	1954-03-09 15:35:26	1961-09-03	1990-09-24	2002-12-25 02:40:12	1954-03-09 15:35:26	1961-09-03	1990-09-24
2	3.0	1973-08-02 03:27:41	1962-11-30 08:56:23	1962-04-25	1966-10-01	1973-08-02 03:27:41	1962-11-30 08:56:23	1962-04-25	1966-10-01
3	4.0	1984-04-08 17:06:49	1935-04-04 15:34:40	1962-11-16	1979-04-14	1984-04-08 17:06:49	1935-04-04 15:34:40	1962-11-16	1979-04-14
4	5.0	2003-02-04 19:42:52	1963-02-18 06:53:54	1962-12-04	2013-03-17	2003-02-04 19:42:52	1963-02-18 06:53:54	1962-12-04	2013-03-17
5	6.0	1966-09-02 09:37:17	1935-11-02 16:23:38	1962-01-10	2000-07-13	1966-09-02 09:37:17	1935-11-02 16:23:38	1962-01-10	2000-07-13
6	7.0	1969-09-26 22:23:10	1964-09-19 22:15:04	1960-06-28	1986-09-18	1969-09-26 22:23:10	1964-09-19 22:15:04	1960-06-28	1986-09-18
7	8.0	1995-02-10 20:17:57	1963-02-08 06:09:53	1964-03-01	1984-05-10	1995-02-10 20:17:57	1963-02-08 06:09:53	1964-03-01	1984-05-10
8	9.0	1970-09-01 17:19:39	1970-09-01 13:58:32	1960-12-20	2018-01-05	1970-09-01 17:19:39	1970-09-01 13:58:32	1960-12-20	2018-01-05
9	10.0	1979-12-20 04:57:52	1963-06-22 05:37:12	1963-01-21	1985-04-07	1979-12-20 04:57:52	1963-06-22 05:37:12	1963-01-21	1985-04-07
10	11.0	2002-07-19 18:23:49	NaT	1960-10-02	1960-08-21	2002-07-19 18:23:49	NaT	1960-10-02	1960-08-21
11	12.0	2001-09-04 03:25:32	1961-09-28 03:25:32	1963-09-19	1963-09-19	2001-09-04 03:25:32	1961-09-28 03:25:32	1963-09-19	1963-09-19

We can use Pandas read_spss to read an Spss dataset into a Python data frame. But, that demands that pyreadstat is installed and imported:

```
!pip install pyreadstat
import pyreadstat
```

When it is installed, it is easy to import:

```
mdgperson_from_spss = pd.read_spss(datapath + 'mdgperson.sav')
```

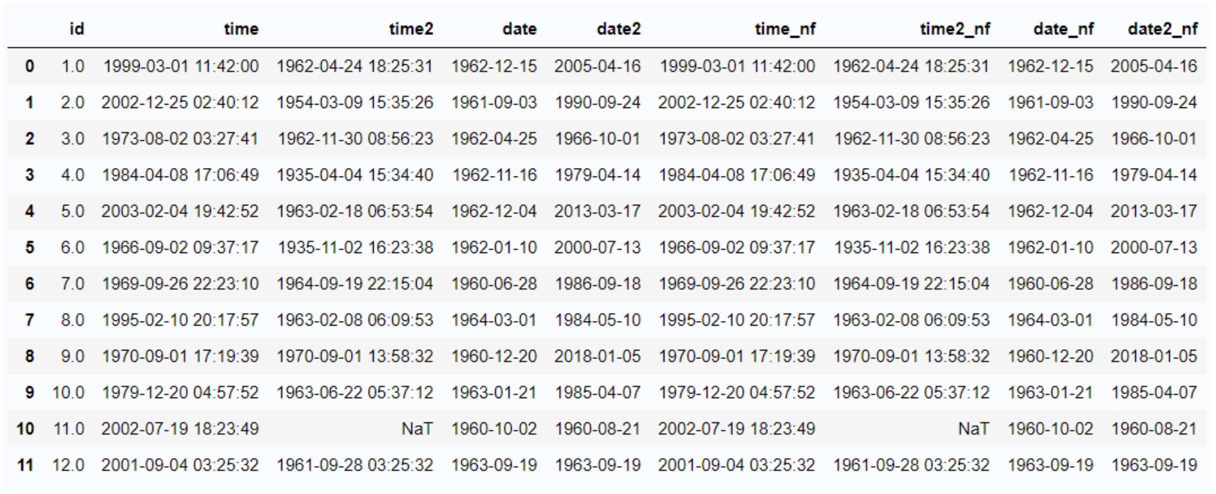
For variables with value labels, these are converted into category variables so that the labels are imported to the Python data frame.

When we import date and time variables, those who are defined as datetime variables in Spss are converted without problems. For date variables, the number behind the actual date is imported. That is the number of seconds since October 14, 1582. We can convert that number to a date number starting from January 1, 1970, which is the Python time 0 by subtracting the number 12219379200. We also tell that the date and time is given in seconds with the parameter *unit*:

```
times_spss = pd.read_spss(datapath + 'times.sav')
times_spss['date']= pd.to_datetime(times_spss['date'] - 12219379200,
unit="s")
times_spss['date2']= pd.to_datetime(times_spss['date2'] - 12219379200,
unit="s")
times_spss['time_nf']= pd.to_datetime(times_spss['time_nf'] - 12219379200,
unit="s")
times_spss['time2_nf']= pd.to_datetime(times_spss['time2_nf'] - 12219379200,
unit="s")
times_spss['date_nf']= pd.to_datetime(times_spss['date_nf'] - 12219379200,
unit="s")
times_spss['date2_nf']= pd.to_datetime(times_spss['date2_nf'] - 12219379200,
unit="s")
times_spss
```

We may not use the origin parameter with a Timestamp as the first valid timestamp is September 22, 1677.

The output has now datetime variables that are converted correctly:



```
      id                 time                time2        date       date2              time_nf             time2_nf     date_nf    date2_nf
0    1.0  1999-03-01 11:42:00  1962-04-24 18:25:31  1962-12-15  2005-04-16  1999-03-01 11:42:00  1962-04-24 18:25:31  1962-12-15  2005-04-16
1    2.0  2002-12-25 02:40:12  1954-03-09 15:35:26  1961-09-03  1990-09-24  2002-12-25 02:40:12  1954-03-09 15:35:26  1961-09-03  1990-09-24
2    3.0  1973-08-02 03:27:41  1962-11-30 08:56:23  1962-04-25  1966-10-01  1973-08-02 03:27:41  1962-11-30 08:56:23  1962-04-25  1966-10-01
3    4.0  1984-04-08 17:06:49  1935-04-04 15:34:40  1962-11-16  1979-04-14  1984-04-08 17:06:49  1935-04-04 15:34:40  1962-11-16  1979-04-14
4    5.0  2003-02-04 19:42:52  1963-02-18 06:53:54  1962-12-04  2013-03-17  2003-02-04 19:42:52  1963-02-18 06:53:54  1962-12-04  2013-03-17
5    6.0  1966-09-02 09:37:17  1935-11-02 16:23:38  1962-01-10  2000-07-13  1966-09-02 09:37:17  1935-11-02 16:23:38  1962-01-10  2000-07-13
6    7.0  1969-09-26 22:23:10  1964-09-19 22:15:04  1960-06-28  1986-09-18  1969-09-26 22:23:10  1964-09-19 22:15:04  1960-06-28  1986-09-18
7    8.0  1995-02-10 20:17:57  1963-02-08 06:09:53  1964-03-01  1984-05-10  1995-02-10 20:17:57  1963-02-08 06:09:53  1964-03-01  1984-05-10
8    9.0  1970-09-01 17:19:39  1970-09-01 13:58:32  1960-12-20  2018-01-05  1970-09-01 17:19:39  1970-09-01 13:58:32  1960-12-20  2018-01-05
9   10.0  1979-12-20 04:57:52  1963-06-22 05:37:12  1963-01-21  1985-04-07  1979-12-20 04:57:52  1963-06-22 05:37:12  1963-01-21  1985-04-07
10  11.0  2002-07-19 18:23:49                  NaT  1960-10-02  1960-08-21  2002-07-19 18:23:49                  NaT  1960-10-02  1960-08-21
11  12.0  2001-09-04 03:25:32  1961-09-28 03:25:32  1963-09-19  1963-09-19  2001-09-04 03:25:32  1961-09-28 03:25:32  1963-09-19  1963-09-19
```

Invalid date and time are given the missing value NaT (Not a Time).

We can use Pandas read_stata to read a Stata dataset into a Python data frame. But that demands that pyreadstat is installed and imported:

```
!pip install pyreadstat
import pyreadstat
```

When it is installed, it is easy to import:

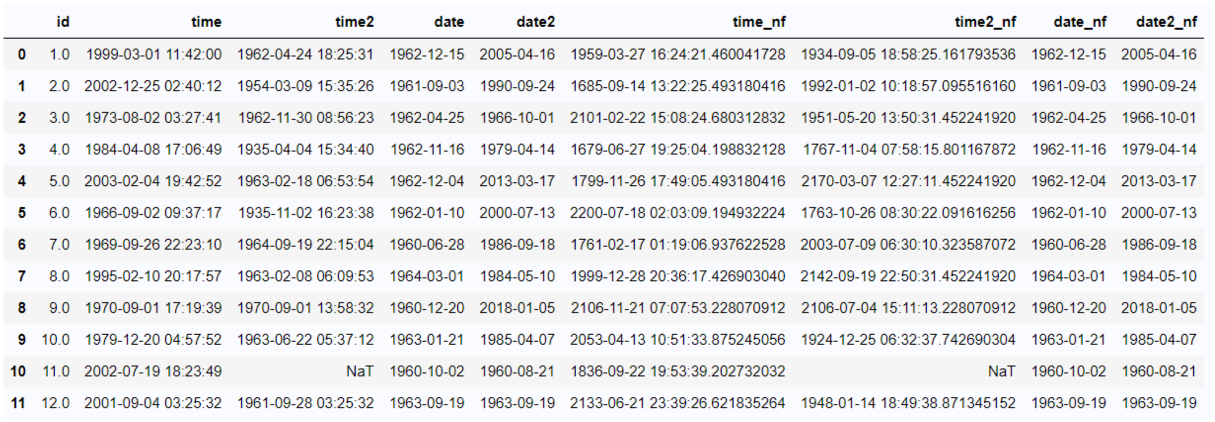
```
mdgperson_from_stata = pd.read_stata(datapath + 'mdgperson.dta')
```

For variables with value labels, these are converted into category variables so that the labels are imported to the Python data frame.

Date and time variables are converted when they are formatted as date or datetime variables. If they are not formatted in Stata, the number of the date or time will be imported. They can be converted with the Pandas *timedelta* function. We add the number of days or seconds since day 0 in Stata, January 1, 1960:

```
times_stata = pd.read_stata(datapath + 'times.dta')
times_stata['time_nf'] = pd.to_timedelta(times_stata['time_nf'], unit='s') +
pd.Timestamp('1960-01-01')
times_stata['time2_nf'] = pd.to_timedelta(times_stata['time2_nf'], unit='s')
+ pd.Timestamp('1960-01-01')
times_stata['date_nf'] = pd.to_timedelta(times_stata['date_nf'], unit='D') +
pd.Timestamp('1960-01-01')
times_stata['date2_nf'] = pd.to_timedelta(times_stata['date2_nf'], unit='D')
+ pd.Timestamp('1960-01-01')
times_stata
```

The Python frame with converted datetime variables:



	id	time	time2	date	date2	time_nf	time2_nf	date_nf	date2_nf
0	1.0	1999-03-01 11:42:00	1962-04-24 18:25:31	1962-12-15	2005-04-16	1959-03-27 16:24:21.460041728	1934-09-05 18:58:25.161793536	1962-12-15	2005-04-16
1	2.0	2002-12-25 02:40:12	1954-03-09 15:35:26	1961-09-03	1990-09-24	1685-09-14 13:22:25.493180416	1992-01-02 10:18:57.095516160	1961-09-03	1990-09-24
2	3.0	1973-08-02 03:27:41	1962-11-30 08:56:23	1962-04-25	1966-10-01	2101-02-22 15:08:24.680312832	1951-05-20 13:50:31.452241920	1962-04-25	1966-10-01
3	4.0	1984-04-08 17:06:49	1935-04-04 15:34:40	1962-11-16	1979-04-14	1679-06-27 19:25:04.198832128	1767-11-04 07:58:15.801167872	1962-11-16	1979-04-14
4	5.0	2003-02-04 19:42:52	1963-02-18 06:53:54	1962-12-04	2013-03-17	1799-11-26 17:49:05.493180416	2170-03-07 12:27:11.452241920	1962-12-04	2013-03-17
5	6.0	1966-09-02 09:37:17	1935-11-02 16:23:38	1962-01-10	2000-07-13	2200-07-18 02:03:09.194932224	1763-10-26 08:30:22.091616256	1962-01-10	2000-07-13
6	7.0	1969-09-26 22:23:10	1964-09-19 22:15:04	1960-06-28	1986-09-18	1761-02-17 01:19:06.937622528	2003-07-09 06:30:10.323587072	1960-06-28	1986-09-18
7	8.0	1995-02-10 20:17:57	1963-02-08 06:09:53	1964-03-01	1984-05-10	1999-12-28 20:36:17.426903040	2142-09-19 22:50:31.452241920	1964-03-01	1984-05-10
8	9.0	1970-09-01 17:19:39	1970-09-01 13:58:32	1960-12-20	2018-01-05	2106-11-21 07:07:53.228070912	2106-07-04 15:11:13.228070912	1960-12-20	2018-01-05
9	10.0	1979-12-20 04:57:52	1963-06-22 05:37:12	1963-01-21	1985-04-07	2053-04-13 10:51:33.875245056	1924-12-25 06:32:37.742690304	1963-01-21	1985-04-07
10	11.0	2002-07-19 18:23:49	NaT	1960-10-02	1960-08-21	1836-09-22 19:53:39.202732032	NaT	1960-10-02	1960-08-21
11	12.0	2001-09-04 03:25:32	1961-09-28 03:25:32	1963-09-19	1963-09-19	2133-06-21 23:39:26.621835264	1948-01-14 18:49:38.871345152	1963-09-19	1963-09-19

There is a Python library called *pyreadr* which can be used to import R data frames to Python. First, we have to install and import the library:

```
!pip install pyreadr
import pyreadr
```

Now we can import an Rdata file. It returns an ordered dictionary which we can extract to a data frame:

```
result = pyreadr.read_r(datapath + 'mdgperson_nodup.Rdata')
mdgperson_from_rdata = result['mdgperson_nodup']
mdgperson_from_rdata
```

To import an rds file, we do almost the same. The only difference is that data imported to result has no name. We use None instead:

```
result = pyreadr.read_r(datapath + 'mdgperson_nodup.rds')
mdgperson_from_rds = result[None]
mdgperson_from_rds
```

Export
There is no direct way in Python to convert to Sas datasets without using the saspy library. The saspy library needs Sas to be installed on our computer. Then we must connect to Sas from Python, and in order to do that we need to some setup first. It is described here: https://sassoftware.github.io/saspy/.

There is a *write_xport* function to export to Sas xport format. However, it does not create an xport file that Sas can read. Instead, we can create a csv file and import it in Sas.

To export a Python data frame to an Spss dataset we can use pyreadstat as well. The variables are exported with their values, which also goes for category variables. That means it is the text for the category that is exported, not the code behind. We can do it like this:

```
pyreadstat.write_sav(mdgperson_nodup, datapath +
'mdgperson_nodup_from_python.sav')
```

When we write a Python data frame to an Spss dataset, datetime variables will also be datetime variables in Spss:

```
pyreadstat.write_sav(times_spss, datapath + 'times_from_python.sav')
```

To export a Python data frame to a Stata dataset we can use *to_stata* function. The variables are exported with their values, which also goes for category variables. That means it is the text for the category that is exported, not the code behind. We can do it like this:

```
mdgperson_nodup.to_stata(datapath + 'mdgperson_nodup_from_python.dta')
```

This syntax is also possible, however it may throw an error message in Stata, depending on the Stata implementation:

```
pyreadstat.write_dta(mdgperson_nodup, datapath +
'mdgperson_nodup_from_python.dta')
```

This message may appear:

```
dataset too large

    This .dta file format was created by Stata/MP and has more variables than your
Stata can handle.
```

When we save a Python data frame to a Stata dataset all datetime variables will be kept as datetime variables in Stata. Syntax like this should work:

```
times_stata.to_stata(datapath + 'times_from_python.dta')
```

There is no method to export to Rdata or rds datasets in base Python. Instead, we can save the data frame as a feather file. To do that we will first have to install the feather-format package:

```
!pip install feather-format
```

Then we will import the package.

```
import feather
```

Finally, we can write to the feather format:

```
feather.write_dataframe(mdgperson_nodup, datapath +
'mdgperson_nodup_from_python.feather')
```

There is a feather library in R. However, when we try to use that on feather files made with Python, we may get an error message like this:

```
Error in openFeather(path) : Invalid: Not a feather file
```

Instead, we can use the arrow package in R. It will read the feather file from Python. The R program may look like this:

```
install.packages('arrow')
library(arrow)
mdgperson_nodup_from_python <- arrow::read_feather('h:/mdg/data/mdgperson_nod
up_from_python.feather')
```

It is by the time of writing not possible to read Python files like pickles into R. However, to import Python files in R it should be possible to call Python from R. To do that we can use a package called reticulate. But then it is a chance that R does not find any Python installations (even though Python is installed!). If it does not find any Python, R will ask if we will install a Miniconda version of Python.

We can do that by answering y to that question. We also need to install Pandas and create a Python script that reads a pickle file. The Python script may look like this:

```
import pandas as pd
def read_pickle_file(file):
    pickle_data = pd.read_pickle(file)
    return pickle_data
```

In R, we call the Python script (called pickle_reader.py) in a program like this:

```
install.packages('reticulate')
require("reticulate")
py_install("pandas")
py_install("pickle")
source_python("h:/mdg/syntax/pickle_reader.py")
nmdgperson_nodup_from_pickle <- read_pickle_file("H:/mdg/data/mdgperson_nodup
.pkl")
```

However, it may give us error messages like this.

```
PackagesNotFoundError: The following packages are not available from current
channels:

  - pickle

Error: one or more Python packages failed to install [error code 1]

Error in py_get_attr_impl(x, name, silent) :

  AttributeError: 'NoneType' object has no attribute 'axes'
```

It is better to save to the feather format in Python because R can read feather files. We can try a program like this:

```
library(feather)
mdgperson_nodup_from_python <- read_feather('h:/mdg/data/mdgperson_nodup_from
_python.feather')
```

An error message like this may appear:

```
Error in openFeather(path) : Invalid: Not a feather file
```

Instead, we can use the arrow package as shown above.

Appendix A:

Command names

These are sets of commonly used commands and subcommands used in Sas, Spss, Stata, R and Python:

Action	Sas	Spss	Stata	R	Python
Define an import dataset	Input	Data list	infix, infile or insheet	read	read
Naming an external import file	Infile	File	using (subcommand)	Within read command	Within read command
Labelling variables	Label	Variable labels	label variable	upData (in Hmisc package)	attrs
Labelling values	Proc format	Value labels	label define	factor with levels and labels arguments	Dictionary with keys and values
Connect value labels to variables	Format	Value labels	label values	factor with levels and labels arguments	map or replace from dictionary
Save dataset	Data	Save outfile	save or saveold	save or saveRDS	pd.to_json, pd.to_pickle, pd.to_parquet and others
One-way frequency table	Proc freq	Frequencies	tab1	table	pd.crosstab or stb.freq (from siedtable module)
Two-way frequency table	Proc freq	Crosstabs	tabulate	table	pd.crosstab
Descriptive statistics	Proc means	Descriptives	tabstat	summary	describe
Descriptive statistics grouped	Proc means	Means	tabstat	summarise in dplyr package	groupby with agg function
Subsets of data	Where or If	Select If, Temporary	if (qualifier), drop	subset	loc
Conditions	If, Where or Select	If or Do if	if (qualifier), recode	if, ifelse, filter in dplyr package	if, np.where
Sort data	Proc sort or Proc sql	Sort cases	sort	order, arrange in dplyr package	sort_values
Read a dataset	Set	Get file	use	read	read_...
List data	Proc print	List	list	print, head, str	print, head, sample
Compute on variables	assignment: variable=expression (no keyword)	Compute	generate (new variable) replace (existing variable) or egen	assignment: variable <- expression, mutate in dplyr package	assignment: variable = expression (no keyword)

Delete variables	Drop or keep	Delete variables	drop or keep	subset	drop
Define working directory	Libname	Cd	cd	setwd	os.chdir()
Match files	Proc sql or Data step with Merge and By	Match files	joinby or merge	full_join, inner_join, left_join, right_join, merge	pd.merge
Aggregate data	Proc sql	Aggregate	collapse	group_by and summarise in dplyr package	groupby with agg function or groupby with transform
Restructure from observations to variables	Proc transpose and Data steps with Merge and By	Casetovars	reshape	reshape	pivot
Restructure from variables to observations	Data step with Set, By, Array, If, Do, Output, Label and Format	Varstocases	reshape	reshape	wide_to_long
Defining an array	Array	Vector	varlist (subcommand to foreach)	array	array
Looping	Do	Loop	foreach	for	for
Recoding	Select	Recode	recode	case_when in dplyr package	cut, np.where or define a function
Format variables	Format	Formats	format	format	format
Tabulation	Proc tabulate or Proc report	Ctables	table, tab2, tabulate	tabular in tables package	pd.crosstab
Graphs	Proc gchart, Proc gplot and other graph procedures	Graph or Ggraph	graph	ggplot2 or plotly	plotly, matplotlib, seaborn and others
Import data files from other software	Proc import	Get Sas or Get Stata	N/A (or download usesas or usespss)	read_sas, read_sav, read_stata from package haven, read.sas7bdat from package sas7bdat	read_sas, read_spss, read_stata, read_r
Export data files to other software	Proc export	Save Translate	outfile (plain text format) or outsheet	write_sas, write_spss, write_stata, from package haven	write_sav, to_stata
Comment	/* comment */ or * comment ;	/* comment */ or * comment .	* comment line // comment rest of line /* comment */	# comment (on each line)	# comment (on each line)

Person dataset

The dataset that is used in most of the examples in this document is listed below. The positions in the dataset are these:

hh	1 - 6	Household identification
state	7 - 7	State
urbrur	8 - 8	Urban/rural location of household
member	9 - 9	Member number within household
b3	10 - 11	Relationship to head of household
b4	12 - 12	Sex
b5	13 - 14	Age
b6	15 - 15	Civil status

This is the dataset with fixed positions which is used in most of the examples:

```
020074215 2110.
020074211 01393
060036614 21201
040024411 02203
040024412 12332
040024412112233
050069525 42161
060036613 22241
020074213 21161
050069522 12602
020118211 01272
060041615 21 8.
020118215 22 3.
020074216 22 8.
060036615 21181
040024414 21141
060036616 21161
020074214 22131
020100213 22211
020118214 22 5.
050069526 41131
040024415102181
040024415 21 9.
020100212 12412
020100216 2110.
060041611 02312
050069521 01672
060041614 22171
060041613101 .1
020118212 12222
020074212 12212
020100215 22161
020118213 22 8.
040024413112 7.
020100214 21192
060036611 01422
020100211 01453
060041612 21201
050069524 22201
040024414102 9.
040024416 2211.
050069523 22302
060036612 12402
060041614 22171
060041616 21 1.
040024416101121
040024411 01372
040024413 22171
```