Data Management Using Stata: A Practical Handbook

Second Edition

MICHAEL N. MITCHELL

A Stata Press Publication
StataCorp LLC
College Station, Texas

Published by Stata Press, 4905 Lakeway Drive, College Station, Texas 77845
Typeset in LaTeX 2ε
Printed in the United States of America
10 9 8 7 6 5 4 3 2 1

Print ISBN-10: 1-59718-318-0
Print ISBN-13: 978-1-59718-318-5
ePub ISBN-10: 1-59718-319-9
ePub ISBN-13: 978-1-59718-319-2
Mobi ISBN-10: 1-59718-320-2
Mobi ISBN-13: 978-1-59718-320-8

Library of Congress Control Number: 2020938361

(Pages omitted)

Contents

(Pages omitted)

Preface to the Second Edition

It was nearly 10 years ago that I wrote the preface for the first edition of this book. The goals and scope of this book are still the same, but in this second edition you will find new data management features that have been added over the last 10 years. Such features include the ability to read and write a wide variety of file formats, the ability to write highly customized Excel files, the ability to have multiple Stata datasets open at once, and the ability to store and manipulate string variables stored as Unicode.

As mentioned above, Stata now reads many file formats. Stata can now read Excel files (see section 2.3), SAS files (see section 2.4), SPSS files (see section 2.5), and even dBase files (see section 2.6). Further, Stata has added the `import delimited` command, which reads a wide variety of delimited files and supports many options for customizing the importing of such data (see section 2.7.1).

Stata can now export files into many file formats. Stata can now export Excel files (see section 3.3), SAS XPORT 8 and SAS XPORT 5 files (see sections 3.4 and 3.5), and dBase files (see section 3.6). Additionally, the `export delimited` command exports delimited files and supports many options for customizing the export of such data (see section 3.7). Also, section 3.9 will illustrate some of the enhanced capabilities Stata now has for exporting Excel files, showing how you can generate custom formatted reports.

The biggest change you will find in this new edition is the addition of chapter 11, titled "Programming for data management: Part II". Chapter 11 builds upon chapter 10, illustrating how Stata programs can be used to solve common data management tasks. I describe four strategies that I commonly use when creating a program to solve a data management task and illustrate how to solve 10 different data management problems, drawing upon these strategies as part of solving each problem. The concluding discussions of each example talk about the strategies and programming tools involved in solving the example. I chose the 10 examples in this chapter not only because the problems are common and easy to grasp but also because these programs illustrate frequently used tools for writing Stata programs. After you explore these examples and see these programming tools applied to data management problems, I hope you will have insight into how you can apply these tools to build programs for your own data management tasks.

Writing this book has been both a challenge and a pleasure. I hope that you like it!

Ventura, CA
May 2020

Michael N. Mitchell

(Pages omitted)

6 Creating variables

Not everything that can be counted counts, and not everything that counts can be counted.

—Albert Einstein

6.1 Introduction

This chapter covers many ways that you can create variables in Stata. I start by introducing the `generate` and `replace` commands for creating new variables and changing the contents of existing variables (see section 6.2). The next two sections describe how you can use numeric expressions and functions when creating variables (see section 6.3) and how you can use string expressions and functions when creating variables (see section 6.4). Section 6.5 illustrates tools to recode variables.

Tools for coding missing values are illustrated in section 6.6, which is followed by a discussion of dummy variables and the broader issue of factor variables (see section 6.7). Section 6.8 covers creating and using date variables, and section 6.9 covers creating and using date-and-time variables.

The next three sections illustrate the use of the `egen` command for computations across variables within each observation (section 6.10), for computations across observations (section 6.11), and for additional functions (section 6.12).

Methods for converting string variables to numeric variables are illustrated in section 6.13, and section 6.14 shows how numeric variables can be converted to string variables.

The chapter concludes with section 6.15, which illustrates how to rename and order variables.

6.2 Creating and changing variables

The two most common commands used for creating and modifying variables are the `generate` and `replace` commands. These commands are identical except that `generate` creates a new variable, while `replace` alters the values of an existing variable. I illustrate these two commands using `wws2.dta`, which contains demographic and labor force information regarding 2,246 women. Consider the variable `wage`, which contains the woman's hourly wages. This variable is summarized below. It has two missing values (the $N = 2244$).

```
. use wws2
(Working Women Survey w/fixes)
. summarize wage
    Variable |        Obs        Mean    Std. Dev.       Min        Max
-------------+---------------------------------------------------------
        wage |      2,244    7.796781     5.82459          0   40.74659
```

Say that we want to compute a weekly wage for these women based on a 40-hour work week. We use the `generate` command to create the new variable, called `wageweek`, which contains the value of `wage` multiplied by 40.

```
. generate wageweek = wage*40
(2 missing values generated)
```

```
. summarize wageweek
```

Variable	Obs	Mean	Std. Dev.	Min	Max
wageweek	2,244	311.8712	232.9836	0	1629.864

This dataset also contains a variable named `hours`, which is the typical number of hours the woman works per week. Let's create `wageweek` again but use `hours` in place of 40. Because `wageweek` already exists, we must use the `replace` command to indicate that we want to replace the contents of the existing variable. Note that because `hours` has 4 missing observations, the `wageweek` variable now has 4 additional missing observations, having only 2,240 valid observations instead of 2,244.[1]

```
. replace wageweek = wage*hours
(1,152 real changes made, 4 to missing)
. summarize wageweek
```

Variable	Obs	Mean	Std. Dev.	Min	Max
wageweek	2,240	300.2539	259.2544	0	1920

Tip! Ordering variables with the generate command

When creating a new variable using the `generate` command, you can use the `before()` or `after()` option to specify where the new variable will be positioned within the dataset. For example, we could have used the `generate` command as follows to create `wageweek`, positioning it after the variable `wage`:

```
. generate wageweek = wage*40, after(wage)
```

The `generate` and `replace` commands can be used together when a variable takes multiple steps to create. Consider the variables `married` (which is 1 if the woman is currently married and 0 otherwise) and `nevermarried` (which is 1 if she was never married and 0 if she is married or was previously married). We can place the women into three groups based on the cross-tabulation of these two variables.

```
. tabulate married nevermarried
```

	Woman never been married		
married	0	1	Total
0	570	234	804
1	1,440	2	1,442
Total	2,010	236	2,246

1. When a variable is missing as part of an arithmetic expression, then the result of the expression is missing.

Say that we want to create a variable that reflects whether a woman is 1) single and has never married ($n = 234$), 2) currently married ($n = 1440$), or 3) single but previously married ($n = 570$). Those who are (nonsensically) currently married and have never been married ($n = 2$) will be assigned a value of missing.[2] This can be done as shown below. The first `generate` command creates the variable `smd` (for single, married, or divorced or widowed) and assigns a value of 1 if the woman meets the criteria for being single (and never married). The `replace` command assigns a value of 2 if the woman meets the criteria for being currently married. The second `replace` command assigns a value of 3 if the woman meets the criteria for being divorced or widowed. The third `replace` command is superfluous but clearly shows that `smd` is missing for those nonsense cases where the woman is currently married and has never been married. (For more information about the use of `if`, see section A.8.)

```
. generate smd = 1 if (married == 0) & (nevermarried == 1)
(2,012 missing values generated)
. replace  smd = 2 if (married == 1) & (nevermarried == 0)
(1,440 real changes made)
. replace  smd = 3 if (married == 0) & (nevermarried == 0)
(570 real changes made)
. replace  smd = . if (married == 1) & (nevermarried == 1)
(0 real changes made)
```

We can double-check this in two ways. First, we can tabulate `smd` and see that the frequencies for `smd` match the frequencies of the two-way table we created above.

```
. tabulate smd, missing
```

smd	Freq.	Percent	Cum.
1	234	10.42	10.42
2	1,440	64.11	74.53
3	570	25.38	99.91
.	2	0.09	100.00
Total	2,246	100.00	

A more direct way to check the creation of this variable is to use the `table` command to make a three-way table of `smd` by `married` by `nevermarried`. As shown below, this also confirms that the values of `smd` properly correspond to the values of `married` and `nevermarried`.

2. The label for this dataset says "with fixes", but clearly not everything was fixed.

```
. table smd married nevermarried
```

	Woman never been married and married			
	0		1	
smd	0	1	0	1
1			234	
2		1,440		
3	570			

We can combine the `generate` and `replace` commands to create a new dummy (0/1) variable based on the values of a continuous variable. For example, let's create a dummy variable called `over40hours` that will be 1 if a woman works over 40 hours and 0 if she works 40 or fewer hours. The `generate` command creates the `over40hours` variable and assigns a value of 0 when the woman works 40 or fewer hours. Then, the `replace` command assigns a value of 1 when the woman works more than 40 hours.

```
. generate over40hours = 0 if (hours <= 40)
(394 missing values generated)
. replace over40hours = 1 if (hours > 40) & !missing(hours)
(390 real changes made)
```

Note that the `replace` command specifies that `over40hours` is 1 if `hours` is over 40 and if `hours` is not missing. Without the second qualifier, people who had missing data on hours would be treated as though they had worked over 40 hours (because missing values are treated as positive infinity). See section A.10 for more on missing values.

We can double-check the creation of this dummy variable with the `tabstat` command, as shown below. When `over40hours` is 0, the value of `hours` ranges from 1 to 40 (as it should); when `over40hours` is 1, the value of `hours` ranges from 41 to 80.

```
. tabstat hours, by(over40hours) statistics(min max)
Summary for variables: hours
     by categories of: over40hours
```

over40hours	min	max
0	1	40
1	41	80
Total	1	80

We can combine these `generate` and `replace` commands into one `generate` command. This can save computation time (because Stata needs to execute only one command) and save you time (because you need to type only one command). This strategy is based on the values a logical expression assumes when it is true or false. When a logical expression is false, it takes on a value of 0; when it is true, it takes on a value of 1. From the previous example, the expression `(hours > 40)` would be 0 (false) when a woman works 40 or fewer hours and would be 1 (true) if a woman works over 40 hours (or had a missing value for `hours`).

Below, we use this one-step strategy to create `over40hours`. Women who worked 40 or fewer hours get a 0 (because the expression is false), and women who worked more than 40 hours get a 1 (because the expression is true). Women with missing values on hours worked get a missing value because they are excluded based on the `if` qualifier. (See section A.6 for more details about logical expressions and examples.)

```
. generate over40hours = (hours > 40) if !missing(hours)
(4 missing values generated)
```

The `tabstat` results below confirm that this variable was created correctly.

```
. tabstat hours, by(over40hours) statistics(min max)
Summary for variables: hours
     by categories of: over40hours
over40hours |       min       max
------------+--------------------
          0 |         1        40
          1 |        41        80
------------+--------------------
      Total |         1        80
```

For more information, see `help generate` and see the next section, which illustrates how to use numeric expressions and functions to create variables.

6.3 Numeric expressions and functions

In the previous section, we used the `generate` and `replace` commands on simple expressions, such as creating a new variable that equaled `wage*40`. This section illustrates more complex expressions and some useful functions that can be used with the `generate` and `replace` commands.

Stata supports the standard mathematical operators of addition (`+`), subtraction (`-`), multiplication (`*`), division (`/`), and exponentiation (`^`) using the standard rules of the order of operators. Parentheses can be used to override the standard order of operators or to provide clarity. I illustrate these operators below to create a nonsense variable named `nonsense` using `wws2.dta`.

```
. use wws2, clear
(Working Women Survey w/fixes)
. generate nonsense = (age*2 + 10)^2 - (grade/10)
(4 missing values generated)
```

Stata also has many mathematical functions that you can include in your `generate` and `replace` commands. The examples below illustrate the `int()` function (which removes any values after the decimal place), the `round()` function (which rounds a number to the desired number of decimal places), the `ln()` function (which yields the natural log), the `log10()` function (which computes the base-10 logarithm), and `sqrt()` (which computes the square root). The first five values are then listed to show the results of using these functions.

```
. generate intwage = int(wage)
(2 missing values generated)
. generate rndwage = round(wage,1)
(2 missing values generated)
. generate lnwage = ln(wage)
(3 missing values generated)
. generate logwage = log10(wage)
(3 missing values generated)
. generate sqrtwage = sqrt(wage)
(2 missing values generated)
. list wage intwage rndwage lnwage logwage sqrtwage in 1/5
```

	wage	intwage	rndwage	lnwage	logwage	sqrtwage
1.	7.15781	7	7	1.968204	.8547801	2.675408
2.	2.447664	2	2	.8951342	.3887518	1.564501
3.	3.824476	3	4	1.341422	.582572	1.955627
4.	14.32367	14	14	2.661913	1.156054	3.784662
5.	5.517124	5	6	1.707857	.7417127	2.348856

Stata has many functions for creating random variables. For example, you can use the `runiform()` (random uniform) function to create a variable with a random number ranging from 0 to 1. Below, I set the seed of the random-function generator to a number picked from thin air,[3] and then I use the `generate` command to create a new variable, `r`, that is a random number between 0 and 1.

```
. set seed 83271
. generate r = runiform()
. summarize r
```

Variable	Obs	Mean	Std. Dev.	Min	Max
r	2,246	.4922679	.2881426	.0002064	.999861

The `rnormal()` (random normal) function allows us to draw random values from a normal distribution with a mean of 0 and a standard deviation of 1, as illustrated below with the variable `randz`. The variable `randiq` is created, drawn from a normal distribution with a mean of 100 and a standard deviation of 15 (which is the same distribution as some IQ tests).

```
. generate randz = rnormal()
. generate randiq = rnormal(100,15)
. summarize randz randiq
```

Variable	Obs	Mean	Std. Dev.	Min	Max
randz	2,246	-.0078068	1.02353	-3.369846	3.485156
randiq	2,246	100.3909	15.07335	49.11288	161.4213

3. Setting the seed guarantees that we get the same series of random numbers every time we run the commands, making results that use random numbers reproducible.

You can even use the `rchi2()` (random chi-squared) function to create a variable representing a random value from a chi-squared distribution. For example, below I create `randchi2`, which draws random values from a chi-squared distribution with 5 degrees of freedom.

```
. generate randchi2 = rchi2(5)
. summarize randchi2
    Variable |        Obs        Mean    Std. Dev.       Min        Max
-------------+---------------------------------------------------------
    randchi2 |      2,246    4.796946    3.032883    .209906   30.87784
```

This section has illustrated just a handful of the numeric functions that are available in Stata. For more information on functions, see section A.7.

6.4 String expressions and functions

The previous section focused on numeric expressions and functions, while this section focuses on string expressions and functions.

We will use `authors.dta` to illustrate string functions (shown below). We first format `name` so that it is displayed using left-justification (see section 5.8).

```
. use authors
. format name %-17s
. list
```

	id	name
1.	1	Ruth Canaale
2.	2	Y. Don Uflossmore
3.	3	thích nhất hạnh
4.	4	J. Carreño Quiñones
5.	5	Ô Knausgård
6.	6	Don b Iteme
7.	7	isaac O'yerbreath
8.	8	Mike avity
9.	9	ÉMILE ZOLA
10.	10	i William Crown
11.	11	Ott W. Onthurt
12.	12	Olive Tu'Drill
13.	13	björk guðmundsdóttir

Note how the names have some errors and inconsistencies; for example, there is an extra space before Ruth's name. Sometimes, the first letter or initial is in lowercase, and sometimes, periods are omitted after initials. By cleaning up these names, we can see how to work with string expressions and functions in Stata.

There are inconsistencies in the capitalization of the authors' names. Below, I use the `ustrtitle()` function to "titlecase" the names (that is, make the first letter of each word in uppercase). This uses Unicode definitions of what constitutes a word. I use the `ustrlower()` and `ustrupper()` functions to convert the names into all lowercase and all uppercase, respectively, according to the Unicode rules of capitalization.

```
. generate name2 = ustrtitle(name)
. generate lowname = ustrlower(name)
. generate upname = ustrupper(name)
. format name2 lowname upname %-23s
. list name2 lowname upname
```

	name2	lowname	upname
1.	Ruth Canaale	ruth canaale	RUTH CANAALE
2.	Y. Don Uflossmore	y. don uflossmore	Y. DON UFLOSSMORE
3.	Thích Nhất Hạnh	thích nhất hạnh	THÍCH NHẤT HẠNH
4.	J. Carreño Quiñones	j. carreño quiñones	J. CARREÑO QUIÑONES
5.	Ô Knausgård	ô knausgård	Ô KNAUSGÅRD
6.	Don B Iteme	don b iteme	DON B ITEME
7.	Isaac O'yerbreath	isaac o'yerbreath	ISAAC O'YERBREATH
8.	Mike Avity	mike avity	MIKE AVITY
9.	Émile Zola	émile zola	ÉMILE ZOLA
10.	I William Crown	i william crown	I WILLIAM CROWN
11.	Ott W. Onthurt	ott w. onthurt	OTT W. ONTHURT
12.	Olive Tu'drill	olive tu'drill	OLIVE TU'DRILL
13.	Björk Guðmundsdóttir	björk guðmundsdóttir	BJÖRK GUÐMUNDSDÓTTIR

We can trim off the leading blanks, like the one in front of Ruth's name, using the `ustrltrim()` function, like this:

```
. generate name3 = ustrltrim(name2)
```

To see the result of the `ustrltrim()` function, we need to left-justify `name2` and `name3` before we list the results.

```
. format name2 name3 %-17s
. list name name2 name3
```

	name	name2	name3
1.	Ruth Canaale	Ruth Canaale	Ruth Canaale
2.	Y. Don Uflossmore	Y. Don Uflossmore	Y. Don Uflossmore
3.	thích nhất hạnh	Thích Nhất Hạnh	Thích Nhất Hạnh
4.	J. Carreño Quiñones	J. Carreño Quiñones	J. Carreño Quiñones
5.	Ô Knausgård	Ô Knausgård	Ô Knausgård
6.	Don b Iteme	Don B Iteme	Don B Iteme
7.	isaac O'yerbreath	Isaac O'yerbreath	Isaac O'yerbreath
8.	Mike avity	Mike Avity	Mike Avity
9.	ÉMILE ZOLA	Émile Zola	Émile Zola
10.	i William Crown	I William Crown	I William Crown
11.	Ott W. Onthurt	Ott W. Onthurt	Ott W. Onthurt
12.	Olive Tu'Drill	Olive Tu'drill	Olive Tu'drill
13.	björk guðmundsdóttir	Björk Guðmundsdóttir	Björk Guðmundsdóttir

Let's identify the names that start with an initial rather than with a full first name. When you look at those names, their second character is either a period or a space. We need a way to extract a piece of the name, starting with the second character and extracting that one character. The `usubstr()` function used with the `generate` command below does exactly this, creating the variable `secondchar`. Then, the value of `firstinit` gets the value of the logical expression that tests if `secondchar` is a space or a period, yielding a 1 if this expression is true and 0 if false (see section 6.2).

```
. generate secondchar = usubstr(name3,2,1)
. generate firstinit = (secondchar == " " | secondchar == ".")
> if !missing(secondchar)
. list name3 secondchar firstinit, abb(20)
```

	name3	secondchar	firstinit
1.	Ruth Canaale	u	0
2.	Y. Don Uflossmore	.	1
3.	Thích Nhất Hạnh	h	0
4.	J. Carreño Quiñones	.	1
5.	Ô Knausgård		1
6.	Don B Iteme	o	0
7.	Isaac O'yerbreath	s	0
8.	Mike Avity	i	0
9.	Émile Zola	m	0
10.	I William Crown		1
11.	Ott W. Onthurt	t	0
12.	Olive Tu'drill	l	0
13.	Björk Guðmundsdóttir	j	0

We might want to take the full name and break it up into first, middle, and last names. Because some of the authors have only two names, we first need to count the number of names. The Unicode-aware version of this function is called `ustrwordcount()`. This is used to count the number of names, using the word-boundary rules of Unicode strings.

```
. generate namecnt = ustrwordcount(name3)
. list name3 namecnt
```

	name3	namecnt
1.	Ruth Canaale	2
2.	Y. Don Uflossmore	4
3.	Thích Nhất Hạnh	3
4.	J. Carreño Quiñones	4
5.	Ô Knausgård	2
6.	Don B Iteme	3
7.	Isaac O'yerbreath	2
8.	Mike Avity	2
9.	Émile Zola	2
10.	I William Crown	3
11.	Ott W. Onthurt	4
12.	Olive Tu'drill	2
13.	Björk Guðmundsdóttir	2

Note how the `ustrwordcount()` function reports four words in the name of the second author. To help understand this better, I use the `ustrword()` function to extract the first, second, third, and fourth word from `name`. These are called `uname1`, `uname2`, `uname3`, and `uname4`. The `list` command then shows the full name along with the first, second, third, and fourth word of the name.

```
. generate uname1 = ustrword(name3,1)
. generate uname2 = ustrword(name3,2)
. generate uname3 = ustrword(name3,3)
(7 missing values generated)
. generate uname4 = ustrword(name3,4)
(10 missing values generated)
```

```
. list name3 uname1 uname2 uname3 uname4
```

	name3	uname1	uname2	uname3	uname4
1.	Ruth Canaale	Ruth	Canaale		
2.	Y. Don Uflossmore	Y	.	Don	Uflossmore
3.	Thích Nhất Hạnh	Thích	Nhất	Hạnh	
4.	J. Carreño Quiñones	J	.	Carreño	Quiñones
5.	Ô Knausgård	Ô	Knausgård		
6.	Don B Iteme	Don	B	Iteme	
7.	Isaac O'yerbreath	Isaac	O'yerbreath		
8.	Mike Avity	Mike	Avity		
9.	Émile Zola	Émile	Zola		
10.	I William Crown	I	William	Crown	
11.	Ott W. Onthurt	Ott	W	.	Onthurt
12.	Olive Tu'drill	Olive	Tu'drill		
13.	Björk Guðmundsdóttir	Björk	Guðmundsdóttir		

Now, it is clear why author 2 is counted as having four words in the name. According to the Unicode word-boundary rules, the single period is being counted as a separate word.

To handle this, I am going to create a new variable named `name4`, where the `.` has been removed from `name3`. The output of the `list` command below confirms that `name4` is the same as `name3` except for the removal of the periods from the name.

```
. generate name4 = usubinstr(name3,".","",.)
. list name3 name4
```

	name3	name4
1.	Ruth Canaale	Ruth Canaale
2.	Y. Don Uflossmore	Y Don Uflossmore
3.	Thích Nhất Hạnh	Thích Nhất Hạnh
4.	J. Carreño Quiñones	J Carreño Quiñones
5.	Ô Knausgård	Ô Knausgård
6.	Don B Iteme	Don B Iteme
7.	Isaac O'yerbreath	Isaac O'yerbreath
8.	Mike Avity	Mike Avity
9.	Émile Zola	Émile Zola
10.	I William Crown	I William Crown
11.	Ott W. Onthurt	Ott W Onthurt
12.	Olive Tu'drill	Olive Tu'drill
13.	Björk Guðmundsdóttir	Björk Guðmundsdóttir

Now, I am going to use the `replace` command to create a new version of `namecnt` that counts the number of words in this new version of name, `name4`.

```
. replace namecnt = ustrwordcount(name4)
(3 real changes made)
. list name4 namecnt
```

	name4	namecnt
1.	Ruth Canaale	2
2.	Y Don Uflossmore	3
3.	Thích Nhất Hạnh	3
4.	J Carreño Quiñones	3
5.	Ô Knausgård	2
6.	Don B Iteme	3
7.	Isaac O'yerbreath	2
8.	Mike Avity	2
9.	Émile Zola	2
10.	I William Crown	3
11.	Ott W Onthurt	3
12.	Olive Tu'drill	2
13.	Björk Guðmundsdóttir	2

The count of the number of names matches what I would expect.

Now, we can split `name4` into first, middle, and last names using the `ustrword()` function. The first name is the first word shown in `name4` (that is, `ustrword(name4,1)`). The second name is the second word if there are three words in `name4` (that is, `ustrword(name4,2) if namecnt == 3`). The last name is based on the number of names the dentist has (that is, `ustrword(name4,namecnt)`).

```
. generate fname = ustrword(name4,1)
. generate mname = ustrword(name4,2) if namecnt == 3
(7 missing values generated)
. generate lname = ustrword(name4,namecnt)
```

Now, I format the first, middle, and last names using a width of 15 with left-justification and then list the first, middle, and last names:

```
. format fname mname lname %-15s
. list name4 fname mname lname
```

	name4	fname	mname	lname
1.	Ruth Canaale	Ruth		Canaale
2.	Y Don Uflossmore	Y	Don	Uflossmore
3.	Thích Nhất Hạnh	Thích	Nhất	Hạnh
4.	J Carreño Quiñones	J	Carreño	Quiñones
5.	Ô Knausgård	Ô		Knausgård
6.	Don B Iteme	Don	B	Iteme
7.	Isaac O'yerbreath	Isaac		O'yerbreath
8.	Mike Avity	Mike		Avity
9.	Émile Zola	Émile		Zola
10.	I William Crown	I	William	Crown
11.	Ott W Onthurt	Ott	W	Onthurt
12.	Olive Tu'drill	Olive		Tu'drill
13.	Björk Guðmundsdóttir	Björk		Guðmundsdóttir

If you look at the values of `fname` and `mname` above, you can see that some of the names are composed of one initial. In every instance, the initial does not have a period after it (because we removed it).

Let's make all the initials have a period after them. In the first `replace` command below, the `ustrlen()` function is used to identify observations where the first name is one character. In such instances, the `fname` variable is replaced with `fname` with a period appended to it (showing that the plus sign can be used to combine strings together). The same strategy is applied to the middle names in the next `replace` command.

```
. replace fname = fname + "." if ustrlen(fname) == 1
(4 real changes made)
. replace mname = mname + "." if ustrlen(mname) == 1
(2 real changes made)
```

Below, we see that the first and middle names always have a period after them if they are one initial.

```
. list fname mname
```

	fname	mname
1.	Ruth	
2.	Y.	Don
3.	Thích	Nhất
4.	J.	Carreño
5.	Ô.	
6.	Don	B.
7.	Isaac	
8.	Mike	
9.	Émile	
10.	I.	William
11.	Ott	W.
12.	Olive	
13.	Björk	

Now that we have repaired the first and middle names, we can join the first, middle, and last names together to form a full name. I then use the `format` command to left-justify the full name.

```
. generate fullname = fname + " " + lname if namecnt == 2
(6 missing values generated)
. replace fullname = fname + " " + mname + " " + lname if namecnt == 3
(6 real changes made)
. format fullname %-30s
```

The output of the `list` command below displays the first, middle, and last names as well as the full name.

```
. list fname mname lname fullname
```

	fname	mname	lname	fullname
1.	Ruth		Canaale	Ruth Canaale
2.	Y.	Don	Uflossmore	Y. Don Uflossmore
3.	Thích	Nhất	Hạnh	Thích Nhất Hạnh
4.	J.	Carreño	Quiñones	J. Carreño Quiñones
5.	Ô.		Knausgård	Ô. Knausgård
6.	Don	B.	Iteme	Don B. Iteme
7.	Isaac		O'yerbreath	Isaac O'yerbreath
8.	Mike		Avity	Mike Avity
9.	Émile		Zola	Émile Zola
10.	I.	William	Crown	I. William Crown
11.	Ott	W.	Onthurt	Ott W. Onthurt
12.	Olive		Tu'drill	Olive Tu'drill
13.	Björk		Guðmundsdóttir	Björk Guðmundsdóttir

The output of the `list` command below shows only the original name and the version of the name we cleaned up.

```
. list name fullname
```

	name	fullname
1.	Ruth Canaale	Ruth Canaale
2.	Y. Don Uflossmore	Y. Don Uflossmore
3.	thích nhất hạnh	Thích Nhất Hạnh
4.	J. Carreño Quiñones	J. Carreño Quiñones
5.	Ô Knausgård	Ô. Knausgård
6.	Don b Iteme	Don B. Iteme
7.	isaac O'yerbreath	Isaac O'yerbreath
8.	Mike avity	Mike Avity
9.	ÉMILE ZOLA	Émile Zola
10.	i William Crown	I. William Crown
11.	Ott W. Onthurt	Ott W. Onthurt
12.	Olive Tu'Drill	Olive Tu'drill
13.	björk guðmundsdóttir	Björk Guðmundsdóttir

For more information about string functions, see `help string functions`. For more information about Unicode, see `help unicode`.

Tip! Long strings

Do you work with datasets with long strings? Stata has a special string variable type called `strL` (pronounced "sturl"). This variable type can be more frugal than a standard string variable, and it can hold large strings, up to 2-billion bytes. You can get a quick overview of long strings by visiting the Stata video tutorial "Tour of long strings and BLOBs in Stata" by searching for "Stata video blobs" with your favorite web browser and search engine.

6.5 Recoding

Sometimes, you want to recode the values of an existing variable to make a new variable, mapping the existing values for the existing variable to new values for the new variable. For example, consider the variable `occupation` from `wws2lab.dta`.

```
. use wws2lab
(Working Women Survey w/fixes)
. codebook occupation, tabulate(20)
```

```
occupation                                                      occupation
```

```
                 type:  numeric (byte)
                label:  occlbl
                range:  [1,13]                       units:  1
        unique values:  13                       missing .:  9/2,246
           tabulation:  Freq.   Numeric  Label
                          319         1  Professional/technical
                          264         2  Managers/admin
                          725         3  Sales
                          101         4  Clerical/unskilled
                           53         5  Craftsmen
                          246         6  Operatives
                           28         7  Transport
                          286         8  Laborers
                            1         9  Farmers
                            9        10  Farm laborers
                           16        11  Service
                            2        12  Household workers
                          187        13  Other
                            9         .
```

Let's recode `occupation` into three categories: white collar, blue collar, and other. Say that we decide that occupations 1–3 will be white collar, 5–8 will be blue collar, and 4 and 9–13 will be other. We recode the variable below, creating a new variable called `occ3`.

```
. recode occupation (1/3=1) (5/8=2) (4 9/13=3), generate(occ3)
(1918 differences between occupation and occ3)
```

We use the `table` command to double check that the variable `occ` was properly recoded into `occ3`.

```
. table occupation occc3
```

occupation	RECODE of occupation (occupation) 1	2	3
Professional/technical	319		
Managers/admin	264		
Sales	725		
Clerical/unskilled			101
Craftsmen		53	
Operatives		246	
Transport		28	
Laborers		286	
Farmers			1
Farm laborers			9
Service			16
Household workers			2
Other			187

This is pretty handy, but it would be nice if the values of `occ3` were labeled. Although we could use the `label define` and `label values` commands to label the values of `occ3` (as illustrated in section 5.4), the example below shows a shortcut that labels the values as part of the recoding process. Value labels are given after the new values in the `recode` command. (Continuation comments are used to make this long command more readable; see section A.4 for more information.)

```
. drop occ3
. recode occupation (1/3=1 "White Collar") ///
>                   (5/8=2 "Blue Collar") ///
>                   (4 9/13=3 "Other"), generate(occ3)
(1918 differences between occupation and occ3)
. label variable occ3 "Occupation in 3 groups"
. table occupation occ3
```

occupation	Occupation in 3 groups White Collar	Blue Collar	Other
Professional/technical	319		
Managers/admin	264		
Sales	725		
Clerical/unskilled			101
Craftsmen		53	
Operatives		246	
Transport		28	
Laborers		286	
Farmers			1
Farm laborers			9
Service			16
Household workers			2
Other			187

The `recode` command can also be useful when applied to continuous variables. Say that we wanted to recode the woman's hourly wage (`wage`) into four categories using the following rules: 0 up to 10 would be coded 1, over 10 to 20 would be coded 2, over 20 to 30 would be coded 3, and over 30 would be coded 4. We can do this as shown below. When you specify `recode` *#1/#2*, all values between *#1* and *#2*, including the boundaries *#1* and *#2* are included. So when we specify `recode wage (0/10=1) (10/20=2)`, 10 is included in both of these rules. In such cases, the first rule encountered takes precedence, so 10 is recoded to having a value of 1.

```
. recode wage (0/10  =1 "0   to 10") ///
>             (10/20 =2 ">10 to 20") ///
>             (20/30 =3 ">20 to 30") ///
>             (30/max=4 ">30 and up"), generate(wage4)
(2244 differences between wage and wage4)
```

We can check this using the `tabstat` command below (see section 4.5). The results confirm that `wage4` was created correctly. For example, for category 2 (over 10 up to 20), the minimum is slightly larger than 10 and the maximum is 20.

```
. tabstat wage, by(wage4) stat(min max)

Summary for variables: wage
     by categories of: wage4 (RECODE of wage (hourly wage))

     wage4 |       min       max
-----------+--------------------
 0   to 10 |         0        10
 >10 to 20 |  10.00805        20
 >20 to 30 |  20.12883        30
>30 and up |  30.19324  40.74659
-----------+--------------------
     Total |         0  40.74659
-----------------------------------
```

We might want to use a rule that 0 up to (but not including) 10 would be coded 1, 10 up to (but not including) 20 would be coded 2, 20 up to (but not including) 30 would be coded 3, and 30 and over would be coded 4. By switching the order of the rules, for example, we can move 10 to belong to category 2 because that rule appears first.

```
. recode wage (30/max=4 "30 and up") ///
>             (20/30 =3 "20 to <30") ///
>             (10/20 =2 "10 to <20") ///
>             (0/10  =1 "0  to <10"), generate(wage4a)
(2244 differences between wage and wage4a)
```

The results of the `tabstat` command below confirm that `wage4a` was recoded properly.

```
. tabstat wage, by(wage4a) stat(min max)
Summary for variables: wage
     by categories of: wage4a (RECODE of wage (hourly wage))
   wage4a |       min       max
----------+--------------------
0  to <10 |         0  9.999998
10 to <20 |        10  19.91143
20 to <30 |        20  29.72623
30 and up |        30  40.74659
----------+--------------------
    Total |         0  40.74659
------------------------------
```

The `recode` command is not the only way to recode variables. Stata has several functions that we can also use for recoding. We can use the `irecode()` function to recode a continuous variable into groups based on a series of cutpoints that you supply. For example, below, the wages are cut into four groups based on the cutpoints 10, 20, and 30. Those with wages up to 10 are coded `0`, over 10 up to 20 are coded `1`, over 20 up to 30 are coded `2`, and over 30 are coded `3`.

```
. generate mywage1 = irecode(wage,10,20,30)
(2 missing values generated)
```

The `tabstat` command confirms the recoding of this variable:

```
. tabstat wage, by(mywage1) stat(min max)
Summary for variables: wage
     by categories of: mywage1
 mywage1 |       min       max
---------+--------------------
       0 |         0        10
       1 |  10.00805        20
       2 |  20.12883        30
       3 |  30.19324  40.74659
---------+--------------------
   Total |         0  40.74659
-----------------------------
```

The `autocode()` function recodes continuous variables into equally spaced groups. Below, we recode `wage` to form three equally spaced groups that span from 0 to 42. The groups are numbered according to the highest value in the group, so 14 represents 0 to 14, then 28 represents over 14 to 28, and finally 42 represents over 28 up to 42. The `tabstat` command confirms the recoding.

```
. generate mywage2 = autocode(wage,3,0,42)
(2 missing values generated)
. tabstat wage, by(mywage2) stat(min max n)
Summary for variables: wage
     by categories of: mywage2
```

mywage2	min	max	N
14	0	13.9694	2068
28	14.00966	27.89049	127
42	28.15219	40.74659	49
Total	0	40.74659	2244

Although the `autocode()` function seeks to equalize the spacing of the groups, the `group()` option of the `egen` command seeks to equalize the number of observations in each group. Below, we create `mywage3` using the `group()` option to create three equally sized groups.[4]

```
. egen mywage3 = cut(wage), group(3)
(2 missing values generated)
```

The values of `mywage3` are numbered 0, 1, and 2. The lower and upper limits of `wage` for each group are selected to attempt to equalize the size of the groups, so the values chosen are not round numbers. The `tabstat` command below shows the lower and upper limits of wages for each of the three groups. The first group ranges from 0 to 4.904, the second group ranges from 4.911 to 8.068, and the third group ranges from 8.075 to 40.747.

```
. tabstat wage, by(mywage3) stat(min max n)
Summary for variables: wage
     by categories of: mywage3
```

mywage3	min	max	N
0	0	4.903378	748
1	4.911432	8.067631	748
2	8.075683	40.74659	748
Total	0	40.74659	2244

See `help recode`, `help irecode`, and `help autocode` for more information on recoding.

4. It is also possible to use the `xtile` command to create equally sized groupings. For example, the command `xtile wage3 = wage, nq(3)` creates three equally sized groupings of the variable `wage` storing those groupings as `wage3`.