Lesson 4: Manipulating Strings and Dates

This lesson covers: how to convert string variables to both numeric and labelled numeric variables (and vice versa), introduces some useful commands in manipulating strings (including a little about how to use **egen**), how to format variables, and how to manipulate dates.

First off, let's set the working directory and load in the dataset we saved at the end of the last lesson, or **Lesson_04.dta** if you want a fresh dataset.

4.1 Converting String Variables to Numeric Variables (and Vice Versa)

String Variables to Numeric Variables

String variables that are composed entirely of numbers, like *calories*, can be converted to numeric variables using either the **destring** command or the **real()** function. The **destring** command either generates a new variable with numeric values of the string variable or replaces the variable entirely with numeric values, and is a wrapper for the **real()** function, i.e. **destring** puts some code around **real()**, giving it additional functionality. There's a distinction between commands and functions: while a command is standalone and can often be used without specifying anything other than variables and maybe some options, a function slots inside a command as an expression. In this case, the **real()** function turns what is inside the brackets, be it a variable or anything else, into a numeric value, if it can. There are plenty of functions in Stata, and we'll meet some more soon.

For **destring**, you specify whether you want to **generate** or **replace** as an option, with the following syntax:

```
destring {variable}, {generate({new variable name}) or replace}
```

Creating a new variable is often the safer way to go. That way, you can see if something unexpected happened. If we try it with *calories* now though, we get an error message:

```
destring calories, generate(calories_numeric)
calories contains nonnumeric characters; no generate
```

If the string variable you are trying to convert to numeric contains any nonnumeric characters (letters, symbols etc.), then you need an additional option, **force**. This option is an explicit acknowledgement that you are aware there are nonnumeric characters in the variable that will be lost by forcing the variable to become numeric. In this case, there are lots of "NR" values of *calories*, and these are preventing **destring** from working without using **force**.

We can either deal with this by replacing the "NR" values with a blank string, "", then re-running the **destring** command, or we can use the **force** option. Let's show both:

```
destring calories, generate(calories_numeric) force
```

```
calories contains nonnumeric characters; calories_numeric generated as int
(13 missing values generated)
```

And if we **replace** the "NR" values instead:

```
replace calories = "" if calories == "NR"
(13 real changes made)
destring calories, generate(calories_numeric_2)
calories has all characters numeric; calories_numeric_2 generated as int
(13 missing values generated)
```

In either case, we receive an output that tells us whether *calories* contains nonnumeric characters, and then (if we use the **generate** option) the name and precision of the generated variable, and finally how many missing values were generated. The generated numeric variable is ordered so it's next to the string variable. The precision of variable generated is dependent on the numbers being generated. Here, we only have integers (whole numbers), and so the variable was generated as **int** (short for integer). Whenever Stata creates a new variable or loads one in from an external file (Excel spreadsheet, CSV etc.), it gives it a precision, with more precision taking up more computer memory. Stata handles this automatically and increases the precision as necessary, so you'll likely never notice this happening. If you want to know more though, please read through the *technical note* on *Precision in Stata* below.

Using the **real()** function works in a similar way, though it's more streamlined. To generate a numeric version of *calories* using the real() function, we would type:

```
gen calories_numeric_3 = real(calories)
(13 missing values generated)
count if calories_numeric == calories_numeric_3
  1,000
```

You don't get the warnings and extra information using the real() function that you would get using the **tostring** command, so be wary about nonnumeric characters.

Numeric Variables to String Variables

Converting numbers to strings is usually simpler, since we don't need to worry about precision: when creating string variables, Stata will use as many characters as necessary to make sure no information is lost.

There are at least two ways of turning numbers to strings: the first is to use the **string()** function, and the second is to use the **tostring** command, which is a wrapper for the **string()** function, the same way the **destring** command is a wrapper for the **real()** function. First, let's create a couple of new variables, turning *calories_numeric* back to a string:

```
tostring(calories_numeric), gen(calories_string)
calories_string generated as str4
gen calories_string_2 = string(calories_numeric),
after(calories_string)
```

The **tostring** command tells use it created a **str4** variable, which just means a string variable with a maximum of 4 characters, while the generate **command** gave no output. We included an option on the **generate** command that positioned the *calories_string2* after *calories_string* – this helpful option effectively gives an **order** command with the **generate** command. The variables created by both variables are identical to the *calories* variable, which means we've not lost any data by converting from strings to numbers and back again. The only difference is that the string variables have "." instead of "", i.e. they converted missing numeric values (full stops) to literal full stops instead of blanks. We can replace these with missing strings, or to "NR", or with anything else we like. We'll use more commands to manipulates strings in **Section 4.3**.

We don't need the string versions of *calories* anymore and we're happy with the first numeric version, so let's drop the now redundant variables and rename *calories_numeric*:

```
drop calories calories_numeric_* calories_string*
rename calories_numeric calories
```

Note the use of the asterisk: when specifying variables, you can specify all variables starting with the same string by adding an asterisk, e.g. *calories_numeric_** will include both *calories_numeric_2* and *calories_numeric_3,* and *calories_string** will include both *calories_string* and *calories_string_2*.

Technical Note: Precision in Stata

Stata has 5 different precisions for numeric variables: **byte**, **int**, **long**, **float**, and **double**, representing increasingly precise numbers. If you search the help files for **data types** then you'll find a page with more details about which types of numbers each type of precision can contain. For example, **int** variables can handle whole numbers between -32,767 and 32,740. If you add a value to an **int** variable that's outside this range, Stata will automatically change the precision of the variable to accommodate the new value. The bigger the number or more decimal places a variable has, the more computer memory that variable takes up, so Stata tries to use the variable with the least number of decimal places without losing information.

Very rarely, the way Stata deals with precision can cause problems with *if* statements, such as "if age == 1.1". If you ever run into a situation where it looks like an *if* statement should include or exclude decimal values, then you might have a precision problem, where certain numbers (including 1.1) can't be expressed exactly at **floats**. There shouldn't be any problems with **byte**, **int** or **long** precisions because they are integers, and there shouldn't be any problems with **doubles** because Stata does all its calculations with **double** precision (so 1.1 will match 1.1). There is, to the best of my knowledge, no easy solution to this problem. One possible solution is to create all variables with **double** precision by specifying the precision with **`generate`**:

```
generate double {new variable} = {some expression}
```

This can be annoying to remember and makes your dataset bigger than it usually needs to be. Alternatively, when using *if* statements, you could specify you want the value to **float** precision:

```
{command} {variable(s)} if float{{variable}) == {some number}
```

The last recommendation possibly makes the most sense to me. When we deal with values with one or more decimal places, we are usually using approximations of the value. For example, values of height are always rounded, usually to the nearest centimetre or millimetre, though sometimes metres or feet if you're measuring buildings. Either way, height is fully continuous so always has an infinite number of decimal places, and we round the value because it isn't practical to do otherwise. So, when using *if* statements to say "if this variable is larger than this amount", it makes sense to reduce the amount by 0.5 units of the smallest level of precision you've measured. For example, if you've measured height to the nearest centimetre and want to include everyone taller than 170 centimetres, you could include everyone taller than 169.5 centimetres (0.5 units of the smallest precision: centimetres). If you've measured to the nearest millimetre, then 1700 millimetres would be 1699.5 millimetres. Stata's precision inaccuracies will almost certainly be much smaller than this, and won't be an issue. This also means that if you want an equivalence statement, e.g. "if height == 170", you need to use two inequalities instead, e.g. "if height >= 169.5 & height < 170.5". Again, though, this requires extra thought that is usually unnecessary.

My recommendation is to not worry about this until you run into a problem: if and when you experience this issue, modify the *if* statement using **`float()`** or reduce or increase the value you're using to account for imprecision in your own measurements.

Let's take a look at *height*, *weight* and *bmi* for a practical example. The *height* and *weight* variables are only accurate to one or two decimal places, but if you switch to the spreadsheet view and click on a few of the values, many of them are very slightly above or below the value they are trying to show, for instance the *weight* value for *id* = 1 is actually 61.000004. This could be because at the **float** precision, there is no representation of 61 (but there is for 61.000004), or it could be because of the way I generated *weight* in the original dataset and the way we converted it from pounds to kilograms. Either way, if we wanted to include everyone in a command whose weight was 61 kilograms or below, this person wouldn't be included, and that would be a problem. We can't recreate this variable to **double** precision, because I didn't create it with **double** precision initially, but either of the other tricks above would help with this issue.

Finally, let's give a concrete example of how **floats** and **doubles** are different by generating *bmi* as a **double** rather than a **float**:

```
gen double bmi2 = weight/height^2, a(bmi)
(95 missing values generated)
```

bmi2 has been created with more precision than *bmi*, since we specified we wanted it generated as a **double**. Currently, the *bmi* and *bmi2* variables are formatted differently, so we can't see the difference yet – let's change the formatting to see the differences (we'll cover formatting in **Section 4.4**):

```
format bmi bmi2 %23.0g
```

If you look across enough decimal places in the spreadsheet view, you'll see that *bmi* and *bmi2* are different. However, if we use the **float()** command to force *bmi2* to temporarily be a float for an *if* statement, we can make *bmi* and *bmi2* appear the same:

```
count if bmi == bmi2
  97
count if bmi == float(bmi2)
  1,000
```

Apart from the 95 missing observations and 2 extra values that are (surprisingly) identical, the **float** and **double** version of *bmi* are different, unless we force the **double** version to be a **float**. Identical **float** and **double** values do sometimes crop up, but it's rare.

That's it for this *technical note*. For most people, this issue will never appear, but it's a good idea to bear it in mind if things aren't working quite as expected. Lastly, we don't need *bmi2*, so let's drop it and reformat *bmi*:

```
drop bmi2
format bmi %9.0g
```

4.2 Converting String Variables to Labelled Numeric Variables (and Vice Versa)

String Variables to Labelled Numeric Variables

Categorical variables, such as *sex*, *hair_colour* and *income*, are often initially string variables when loaded into Stata. It can be useful to convert these types of variable to labelled numeric variables, like *accommodation*, *marathon* and *current_smoker*. While string variables can be used for many purposes, they cannot be used in mathematical commands, such as regression. However, labelled numeric variables can be used. Therefore, if you want to run an analysis looking at whether being male or female affects blood pressure, you'll first need to make *sex* into a numeric variable, preferably a labelled numeric variable so you don't need to remember whether 0 or 1 is "male" or "female".

This can be done using *if* statements, as in **Exercise 3.11**. This can be the best solution in many cases, but Stata has a command that will do a similar job with less code: **`encode`**. This command takes a string variable, converts each unique string to a unique number, generates a label of the same variable name, then applies this label to the numeric variable. By default, the label **`encode`** creates will be in alphabetical order, and as such it is sometimes simpler to manually convert string variables into labelled numeric variables, as it gives you more control. Alternatively, you can create a label and make **`encode`** use that label when converting the string variable. The general syntax of **`encode`** is:

```
encode {string variable}, generate({new variable name})
```

Let's **`encode`** *sex* first:

```
encode sex, gen(sex_numeric)
order sex_numeric, a(sex)
```

The command puts the new variable at the end of the variable list, and doesn't have an option to order it, so we've had to order it ourselves. If you go into the spreadsheet view, you can see that *sex* and *sex_numeric* are identical, though the former is a string and the latter a labelled numeric variable. The values showing as "Female" in *sex_numeric* are actually 1, and "Male" are actually 2, as **`encode`** starts at 1 and goes up alphabetically for each unique string. You can find the label created by **`encode`** by typing:

```
label list
```

```
sex_numeric:
           1 Female
           2 Male
accommodation:
```

It is usual in Stata to code binary variables (variables with only two options) as 0 and 1, rather than 1 and 2, as for various commands Stata interprets 0 as different all other numbers, for example in logistic regression. As such, it may make sense here to create a label for sex ourselves and make encode use that instead of creating one itself, though let's make sure we remove *sex_numeric* and its label first:

```
drop sex_numeric
drop label sex_numeric
label define sex 0 "Female" 1 "Male"
encode sex, gen(sex_numeric) label(sex)
order sex_numeric, a(sex)
label variable sex_numeric "Sex 0=Female 1=Male"
```

A quick glance at the spreadsheet view shows us this worked well, but let's make sure with a tabulation:

```
tabulate sex sex_numeric, missing
```

	Sex 0=Female 1=Male			
Sex	Female	Male	.	Total
	0	0	6	6
Female	479	0	0	479
Male	0	515	0	515
Total	479	515	6	1,000

This looks right – even the missing values are correctly coded. Let's check the values of *sex_numeric* too:

```
tabulate sex sex_numeric, missing nolabel
```

	Sex 0=Female 1=Male			
Sex	0	1	.	Total
	0	0	6	6
Female	479	0	0	479
Male	0	515	0	515
Total	479	515	6	1,000

Ok, that looks fine, so let's encode *hair_colour* and *income*. The values and order of the labels for *hair_colour* look like they'll be fine starting at 1 and going up alphabetically, but we'll define a label for *income* first:

```
encode hair_colour, gen(hair_colour_numeric)
order hair_colour_numeric, a(hair_colour)
label variable hair_colour_numeric "Hair colour"

label define income 1 "<18,000" 2 "18,000 to 30,000" 3 "30,000
to 50,000" 4 "50,000+"
encode income, gen(income_numeric) label(income)
```

```
order income_numeric, a(income)
label variable income_numeric "Income (£s)"
```

Be sure to write each value label correctly: **encode** will add any missing values to the end of the label. For example, if you set 1 equal to "<£18,000" rather than "<18,000" in the *income* label, then **encode** would add another value, 5 set equal to "<18,000", and there would be no values of 1 in the labelled numeric variable. It's also worth pointing out that you can tell **encode** what you want a new label to be called, if you want **encode** to create a label for you, by putting an unused label name in the label option: without this option, **encode** will give the newly created label the same name as the variable.

Labelled Numeric Variables to String Variables

We've covered converting String variables to labelled numeric variables, now let's cover the reverse. The opposite command to **encode** is **decode**, and is simple to use:

```
decode {labelled numeric variable}, generate({new variable name})
```

And as an example:

```
decode income_numeric, gen(income_string)
```

This variable is identical to the original *income* variable: we can check by counting how many values are the same between them:

```
count if income_string == income
  1,000
```

All values are the same, meaning we didn't change any of the information by taking a string variable, converting it to a labelled numeric variable, then converting it back to a string.

Actually, I would prefer income to have the pound symbols in the label, so let's take a quick detour back to last lesson, and modify the income label. Now the numeric version of *income* has been generated, we can change the labels at our leisure:

```
label define income 1 "<£18,000" 2 "£18,000 to £30,000" 3 "£30,000 to £50,000" 4 "£50,000+", modify
```

We now have numeric labelled variables for *sex*, *hair_colour* and *income*, so we don't need the original or recreated string variables:

```
drop sex hair_colour income income_string
```

We can also rename the numeric variables. First though, it's worth pointing out that variable

names, like commands, can be shortened down to the point that they are unique: we don't need to type out all of *hair_colour_numeric* once *hair_colour* has been dropped, because no other variables start with "ha". Instead, we can type:

```
rename sex sex
rename hair hair_colour
rename income income
```

To Stata, this means exactly the same thing as if we'd have written out the existing variables in full. You can always use this trick when specifying existing variables, and Stata will give an error if there's any confusion, so feel free to use this whenever you like. The only caveat is that if you're using a do file, then it's usually in your interests to make sure that the do file is readable, so keeping some longer variable names can help in understanding what the code is doing.

It is also worth pointing out that if you think there must be a more efficient way of getting Stata to run multiple similar commands, then you'd be right. Stata can make use of **loops** to make code much more efficient and save a bunch of typing, and we cover that in **Lesson 6**.

4.3 Manipulating Strings

String variables may need processing, just in general but also before they can be turned into labelled numeric variables. Stata has quite a few commands that can help manipulate string variables to make this easier. These are relatively quick and easy commands (except the first one), so we'll run through a few in this section. The commands I'll introduce are ones that I've found most useful, there are undoubtedly others that may be useful, so don't take this as an exhaustive list.

We have two string variables left to process: *favourite_colours* is a list of up to 3 colours in order of preference, and *date_of_bp_measurement* is a date and will need processing to turn it into the Stata date format. We'll cover *date_of_bp_measurement* in a little bit and focus on *favourite_colours* for now.

split

The first thing we might want to do to *favourite_colours* is separate each colour into its own variable. Let's take a quick look at the *favourite_colours* variable:

```
list favourite_colours in 1/5
```

	favourite_colours
1.	Blue; Yellow; Red
2.	Red; Indigo; Violet
3.	Yellow; Blue; Green
4.	Indigo; ; Yellow
5.	Violet; ; Blue

Each colour is separated by a semicolon, and sometimes there are missing colours. To turn *favourite_colours* into three individual colour variables, we'll need to use a semicolon as a **delimiter**. A delimiter is any character that separates elements of a variable or file, for example, commas are the delimiter in a comma-separated value (csv) file. In Excel, you could use the "Text to Columns" function. In Stata, we can use the **`split`** function to do the same thing. The syntax is as follows:

```
split {string variable}, generate({stub}) parse({delimiter(s)})
{notrim} {destring} limit({})
```

The **`split`** command takes a single string variable, and separates it out into as many new variables as necessary, starting with {**`stub`**}1, then {**`stub`**}2 etc., using one or more delimiters you provide it with in the **`parse()`** option. If you don't provide a **`stub`**, i.e. the start of a new variable name that will be used to create all the new variables, then **`split`** will use the string variable name instead. The **`notrim`** option tells split *not* to remove spaces at the beginning and end of the new variables: you can see spaces after each semicolon in

favourite_colours, and these would be removed by default, but not if we put in the **notrim** option. The **destring** option tries to use the **destring** command on the new variables, which will create numeric variables if the resulting variables are completely numeric or if you use the **force** option. The **limit()** option allows you to specify the maximum number of variables that can be created. Overall, **split** is a very helpful command. Let's use it to create 3 new variables, *favourite_colour_1* to *favourite_colour_3*:

```
split favourite_colours, generate(colour_) parse(;)
variables created as string:
colour_1  colour_2  colour_3
```

colour_1	colour_2	colour_3
Blue	Yellow	Red
Red	Indigo	Violet
Yellow	Blue	Green
Indigo		Yellow

There we have it: three variables showing favourite colours: the only problem is that sometimes leading blank spaces aren't removed – we'll deal with them in a bit. For now, let's add variable labels and order them:

```
label variable colour_1 "1st favourite colour"
label variable colour_2 "2nd favourite colour"
label variable colour_3 "3rd favourite colour"
order colour_*, a(favourite_colours)
```

egen: ends()

There may be times when we just want elements of a list, rather than all elements. For example, we may have just wanted the first or last colour, not all of them, and using **split** can be overkill. To do this, we could use the **egen** command (the extensions to **generate** command), which is really a suite of dedicated functions that allow you to **generate** variables with more complex expressions than when using **generate** alone. The general syntax is:

```
egen {new variable} = {egen function}({variable(s) or something else})
```

Each function within **egen** has its own syntax, all of which can be found if you search the help files for **egen**. We'll cover the **ends()** and **concat()** functions in this section and other functions in later sections, and we'll present the syntax each time we cover a new function. The general syntax of the **ends()** function is:

```
egen {new variable} = ends({string variable}),
punct({delimiter}) {trim} {head or tail or last}
```

There's a little bit to unpack here, as there are three options the need explanation. The first is that the delimiter you want to use goes in the **punct()** option. Stata isn't consistent here, which is both annoying and challenging: **parse()** is used in **split**, **delim()** is used in **import** and **punct()** is used in **ends()**, and they all are specifying a delimiter. The second option, **trim**, removes spaces at the start and end of the element. The third option tells Stata which element(s) of the original variable you want to put in the new variable. Using the first observation of *favourite_colours*:

- **head** selects everything **before** the **first** delimiter
    - e.g. "Blue"
- **tail** selects everything **after** the **first** delimiter
    - e.g. "Yellow; Red"
- **last** selects everything **after** the **last** delimiter
    - e.g. "Red"

This means that if you have more than one instance of a delimiter, you'll need to combine these options to move each element into its own variable, unlike with **split** where it automatically creates as many new variables as necessary. If we wanted to pull out the first colour only, we would type:

```
egen first_colour = ends(favourite_colours), punct(;) head trim
(44 missing values generated)
```

Whereas if we wanted to pull out the last colour only, we would type:

```
egen last_colour = ends(favourite_colours), punct(;) last trim
(75 missing values generated)
```

If we wanted to pull out the second colour, then things get a little trickier. We'll show it, as there may be instances where this is useful, though **split** may be the easier option in most cases. First, we need to create a temporary variable that has all the colours *except* the first one:

```
egen x = ends(favourite_colours), punct(;) tail trim
egen middle_colour = ends(x), punct(;) head trim
(17 missing values generated)
```

The temporary variable, *x*, just contains the remaining colours once we take out the first one. Having a quick look in the spreadsheet view, everything looks like it worked as expected. Using ends() can be useful, though once you have more than a couple of delimiters, then things can become complex quite quickly. In any case, we no longer need the variables we just created, so can drop them:

```
drop first_colour middle_colour last_colour x
```

egen: concat()

We've separated out the elements of *favourite_colours*, but we should also learn how to put them back together again. This is pretty close to the inverse of what we've just done, and it's another **egen** command, this time **concat()**, which means concatenate, i.e. join together. The general syntax is:

```
egen {new variable} = concat({variables}), punct({delimiter})
{decode}
```

You can use any variable type in **concat()**: strings are kept the same, numeric variables are converted to strings, and numeric labelled variables can either be kept as their labelled values or their numeric values, depending on whether you add the **decode** option or not. The delimiter you add goes between whichever variables you specify, and the order you specify the variables is the order they will appear in the new variable. To go from the individual colour variables to *favourite_colours*, we would type:

```
egen favourite_colours_2 = concat(colour_1 colour_2 colour_3),
punct(;)
count if favourite_colours  == favourite_colours_2
  1,000
```

It seems *favourite_colours2* is identical to *favourite_colours*, so we've recreated it exactly. Great!

We can also manipulate strings directly without using **concat()**:

```
gen favourite_colours_3 = colour_1 + ";" + colour_2 + ";" +
colour_3
count if favourite_colours == favourite_colours_3
  1,000
```

The plus sign adds strings together sequentially. You can also use multiplication signs to duplicate strings.

Moving on, though, we don't need any of the *favourite_colours* variables so they can be removed:

```
drop favourite*
```

Other useful string functions

lower(), upper(), proper()

In **Lesson 3**, we capitalised "male" and "female" separately. However, we really just needed all the values to either be capitalised or not, so we could have made everything lower case:

```
replace colour_1 = lower(colour_1)
(956 real changes made)
tab colour_1
```

1st favourite colour	Freq.	Percent	Cum.
blue	130	13.60	13.60
green	149	15.59	29.18
indigo	143	14.96	44.14
orange	143	14.96	59.10
red	123	12.87	71.97
violet	125	13.08	85.04
yellow	143	14.96	100.00
Total	956	100.00	

Equally, we could have made everything upper case:

```
replace colour_1 = upper(colour_1)
(956 real changes made)
tab colour_1
```

1st favourite colour	Freq.	Percent	Cum.
BLUE	130	13.60	13.60
GREEN	149	15.59	29.18
INDIGO	143	14.96	44.14
ORANGE	143	14.96	59.10
RED	123	12.87	71.97
VIOLET	125	13.08	85.04
YELLOW	143	14.96	100.00
Total	956	100.00	

Or we could have capitalised every word in the string:

```
replace colour_1 = proper(colour_1)
(956 real changes made)
tab colour_1
```

1st favourite colour	Freq.	Percent	Cum.
Blue	130	13.60	13.60
Green	149	15.59	29.18
Indigo	143	14.96	44.14
Orange	143	14.96	59.10
Red	123	12.87	71.97
Violet	125	13.08	85.04
Yellow	143	14.96	100.00
Total	956	100.00	

length()

We may want to know how many characters make up a string. I've used this a few times for various purposes, and we can do this with the **`length()`** function:

```
gen x = length(colour_1)
tab x
```

x	Freq.	Percent	Cum.
0	44	4.40	4.40
3	123	12.30	16.70
4	130	13.00	29.70
5	149	14.90	44.60
6	554	55.40	100.00
Total	1,000	100.00	

```
drop x
```

You'll notice that missing values of *colour_1* have lengths of 0, as they are simply a blank string. You can use **`length()`** with variables, but also with regular strings. For instance, if we wanted to find out how many characters make up this sentence:

```
display length("For instance, if we wanted to find out how many characters make up this sentence:")
81
```

The **`display`** (short: **`dis`**) command is useful to know about, as it turns Stata into a calculator that can also do fancy things, like count the number of characters in a string.

strpos()

We may also want to find the (first) location of a specified string within a larger string, which we can do with the **strpos()** function, I think short for "string position". Let's say we wanted to know the (first) location of the letter "e" in the *colour_1* variable, we would give the **strpos()** function the variable (or string) and then the string we want to try to find:

```
gen x = strpos(colour_1,"e")
tab colour_1 x, miss
```

1st favourite colour	x						Total
	0	2	3	4	5	6	
	44	0	0	0	0	0	44
Blue	0	0	0	130	0	0	130
Green	0	0	149	0	0	0	149
Indigo	143	0	0	0	0	0	143
Orange	0	0	0	0	0	143	143
Red	0	123	0	0	0	0	123
Violet	0	0	0	0	125	0	125
Yellow	0	143	0	0	0	0	143
Total	187	266	149	130	125	143	1,000

You can see that the **strpos()** function gives the relative position of the first instance of the specified string, in this case "e". For colours without an "e", **strpos()** returns a result of 0, like with "Indigo" and the missing strings. The positions of each character in Stata start at 1, rather than 0 as in other programming languages. Like **length()**, we can use **strpos()** to find the position of a string within any string:

```
dis strpos("this is a sentence","sen")
11
```

Here, the string "sen" starts at the 11th character of the string "this is a sentence".

substr()

The **substr()** function allows you to select a part of a string based on character positions. To make **substr()** work, you give it a variable (or string), then the character position you want to start at, then how many characters we want to go for. For instance, if we wanted to create a variable that started at the first character, and was 3 characters long, we would type:

```
gen x2 = substr(colour_1,1,3)
tab x2
```

x2	Freq.	Percent	Cum.
Blu	130	13.60	13.60
Gre	149	15.59	29.18
Ind	143	14.96	44.14
Ora	143	14.96	59.10
Red	123	12.87	71.97
Vio	125	13.08	85.04
Yel	143	14.96	100.00
Total	956	100.00	

You can see that all colours have been reduced to their first 3 characters, starting at the first character. We can also specify character positions by how far from the end of the string they are, as when we specified ranges in **Lesson 3**:

```
replace x2 = substr(colour_1,-3,3)
(956 real changes made)
tab x2
```

x2	Freq.	Percent	Cum.
Red	123	12.87	12.87
een	149	15.59	28.45
igo	143	14.96	43.41
let	125	13.08	56.49
low	143	14.96	71.44
lue	130	13.60	85.04
nge	143	14.96	100.00
Total	956	100.00	

Here, you can see that the colours have been reduced to their final 3 characters: only “Red” is the same, and that’s just because it’s 3 letters long.

The character positions don’t need to be fixed numbers: if we wanted to pull out the parts of the colours from the first letter up to the position of the letter “e”, we can (remember we didn’t drop the variable *x*, which is the character position of the first letter “e”):

```
replace x2 = substr(colour_1,1,x)
tab x2
```

x2	Freq.	Percent	Cum.
Blue	130	15.99	15.99
Gre	149	18.33	34.32
Orange	143	17.59	51.91
Re	123	15.13	67.04
Viole	125	15.38	82.41
Ye	143	17.59	100.00
Total	813	100.00	

```
drop x x2
```

Here, you can see that each colour now just goes up to the first letter "e" encountered. There is no "Indigo" colours now, because "Indigo" doesn't contain the letter "e", so has now been set to missing.

You can also nest functions inside each other. Instead of using the variable *x* above, we could have typed:

```
gen x2 = substr(colour_1,1,strpos(colour_1,"e"))
tab x2
```

x2	Freq.	Percent	Cum.
Blue	130	15.99	15.99
Gre	149	18.33	34.32
Orange	143	17.59	51.91
Re	123	15.13	67.04
Viole	125	15.38	82.41
Ye	143	17.59	100.00
Total	813	100.00	

```
drop x2
```

This works for most functions, so you don't need to create intermediate or temporary variables unless you want to.

subinstr()

The final function useful for manipulating strings is one that replaces part of a string with a different string: **`subinstr()`**. This can be extremely useful, though some caution is warranted here: once the string has been modified, then returning it back to normal may be impossible. To make this function work, we give it, 1) a variable (or string), 2) a string that needs replacing, 3) a string to replace it with, and 4) the number of instances of the string to replace within the same observation (unless you have a specific number in mind, the usual thing to do it put a full stop at the end, which means all instances of the string are replaced).

Essentially, **subinstr()** is the "find and replace" of Stata. As an example, let's replace the forward slashes in *date_of_bp_measurement* with dashes:

```
replace date_of_bp_measurement =
subinstr(date_of_bp_measurement,"/","-",.)
(891 real changes made)
list date_of_bp_measurement in 1/5
```

	date~ement
1.	03-12-2018
2.	31-12-2018
3.	24-09-2018
4.	21-07-2017
5.	24-08-2019

I was going to suggest we put back the forward slashes, but I actually like it more with dashes, so we'll leave *date_of_bp_measurement* as it is. We can also get rid of all spaces in the *colour_2* and *colour_3* variables using **subinstr()**. Let's just do it for *colour_2* for now:

```
replace colour_2 = subinstr(colour_2," ","",.)
(1,000 real changes made)
```

Clearly, every observation in *colour_2* had a space in it. All spaces would be removed using this method: if you only wanted to remove spaces at the start and end of an observation (or any string), then you could use the strtrim() function:

```
replace colour_3 = strtrim(colour_3)
(925 real changes made)
```

This time, not all observations had spaces, just those with non-missing values. However, the same job was accomplished – the spaces are gone, which means we can **encode** the colours (before, the spaces may have caused problems with encoding):

```
encode colour_1, gen(colour_1x) label(colours)
encode colour_2, gen(colour_2x) label(colours)
encode colour_3, gen(colour_3x) label(colours)
order colour_1x-colour_3x, a(colour_3)
drop colour_1-colour_3
rename colour_1x colour_1
rename colour_2x colour_2
rename colour_3x colour_3
```

4.4 Formatting Variables

Variables in Stata can be formatted to look how you want them to look. The underlying data doesn't change, but the appearance of values can change markedly. This is similar to applying value labels, but much more general. The syntax for formatting variables is:

```
format {variable(s)} {formatting code}
```

There's lots of different formats, each with their own code, and the help page for **format** is the best place to go to see all of them. If we want to look at how the variables in memory are formatted currently, we can type:

```
format *
```

variable name	display format
id	%9.0g
age	%9.0g
sex	%8.0g
height	%9.0g
weight	%9.0g
bmi	%9.0g
hair_colour	%8.0g
systolic_bp	%9.0g
diastolic_bp	%9.0g
average_bp	%9.0g
accommodation	%9.0g
exercise	%9.0g
calories	%10.0g
income	%20.0g
marathon	%9.0g
current_smoker	%14.0g
colour_1	%8.0g
colour_2	%8.0g
colour_3	%8.0g
date_of_bp_measurement	%10s
date_of_recruitment	%td

Formatting Numeric Variables

The default formatting type for numeric variables is right-justified (touching the right hand side of the cell) general formatting, which has the code:

```
%{number}.{number}g
```

You can see that all variables down to *colour_3* have general formatting, the only difference is

the first number. In general formatting, the first number determines the overall width of the cell in the spreadsheet view, and the second number tells Stata how many numbers should be shown: a value of 0 here leaves it up to Stata how many numbers to show, rather than showing no numbers. This only affects numbers after the decimal place, so integers will still be shown as normal up to the decimal place, but the specification is for the total number of numbers displayed.

For example, if we **`format`** *bmi*:

```
format bmi %9.3g
list bmi in 1/5
```

	bmi
1.	22.1
2.	27.8
3.	26.8
4.	35.1
5.	27

We see that a maximum of 3 digits are displayed. If we do the same to *height*:

```
format height %9.3g
list height in 1/5
```

	height
1.	1.66
2.	1.67
3.	1.81
4.	1.56
5.	1.67

We again see that a maximum of 3 digits are displayed, but because *height* is a lot smaller than *bmi*, we see more numbers after the decimal place. Now if we do the same to *calories*:

```
format calories %9.3g
list calories in 1/5
```

	calories
1.	1605
2.	2329
3.	2124
4.	2301
5.	1960

Although we've specified seeing only 3 numbers, Stata doesn't round anything to the left of the decimal place (integers), it still shows everything. However, if there were numbers after the decimal place in *calories*, they wouldn't be shown unless we had specified at least 5 after the decimal place in the formatting code.

The default formatting of variables is usually fine, but you may prefer to cap the number of numbers that appear after the decimal place without worrying about the number of numbers that appear before the decimal place. For this, we could use the fixed format, where the first number determines the overall width of the cell as before, but the second number specifies the number of numbers to show after the decimal place. For example, if we wanted to cap *weight* at 1 decimal place:

```
format weight %12.1f
list weight in 1/5
```

	weight
1.	61.0
2.	77.6
3.	87.9
4.	85.5
5.	75.4

Fixed formats force the variable to show the specified number of numbers after the decimal place, even if the value is an integer, e.g. "61.0". This can be particularly useful when getting data ready to copy into a table, and you want a specific number of decimal places to show.

For the most part, we can think of general formats as restricting to a number of significant figures (after the decimal place), whereas the fixed formats restrict to a number of decimal places.

For very large or very small numbers, we may want to use the exponential format (called "Scientific" in Excel), e.g. 5.1*10^3 to represent 5100, or 3.7*10^-3 to represent 0.0037. We can use the exponential format for this, which has the same specification as the fixed format, but with an "e" instead of an "f":

```
format calories %9.2e
list calories in 1/5
```

	calories
1.	1.61e+03
2.	2.33e+03
3.	2.12e+03
4.	2.30e+03
5.	1.96e+03

This isn't particularly helpful for *calories*, so let's change it back:

```
format calories %9.0g
```

There are other formats for numeric variables, but they aren't typically used. You can, however make Stata display decimal places are commas instead of full stops, either for all variables using the **set dp** command, or for specific variables by changing the full stop in the format commands to a comma:

```
set dp comma, permanently
list height in 1/5
```

	height
1.	1,66
2.	1,67
3.	1,81
4.	1,56
5.	1,67

```
set dp period, permanently
list height in 1/5
```

	height
1.	1.66
2.	1.67
3.	1.81
4.	1.56
5.	1.67

```
format height %9,0g
list height in 1/5
```

	height
1.	1,66
2.	1,67
3.	1,81
4.	1,56
5.	1,67

```
format height %9.0g
list height in 1/5
```

	height
1.	1.66
2.	1.67
3.	1.81
4.	1.56
5.	1.67

You can left justify variables by adding a minus sign after the % symbol in format commands, for example:

```
format height %-9.0g
list height in 1/5
```

	height
1.	1.66
2.	1.67
3.	1.81
4.	1.56
5.	1.67

```
format height %9.0g
```

That's about it for numeric variables.

Formatting String Variables

String variables are pretty simple, as the format is:

```
format %{number}s
```

The number is the number of characters to show. You can left justify as above, and you can also centre justify by adding a tilde (~) after the % symbol.

Variables representing dates and times have special formatting: you can see the *date_of_recruitment* has the %td formatting, which the general date format. More about dates in the next section.

One final comment on formatting: when you convert numeric variables to strings with the **`string()`** function or the **`tostring`** command, you can select a format for the numbers. For example, if we wanted to create a string variable that showed *bmi* to 1 decimal place, we could type:

```
gen x = string(bmi,"%9.1f")
```

The equivalent command using tostring would be:

```
tostring bmi, gen(x2) format("%9.1f") force
```

```
x2 generated as str4
x2 was forced to string; some loss of information
```

This works in exactly the same was as using the **`string()`** function, but it requires more typing to achieve the same outcome. Because the *x* variable is a string, we can add units to it, using the **`concat()`** function of **`egen`** (after creating a variable with the units):

```
gen units = "kg/m2"
egen x3 = concat(x units), punct(" ")
```

This works well, but missing values of *bmi* now have a value of ". kg/m2" in the new variable. Sadly, **`egen`** doesn't have a **`replace`** equivalent, so we'll drop *x3* and recreate it, but only for non-missing values of *bmi*:

```
drop x3
egen x3 = concat(x units) if bmi < ., punct(" ")
(95 missing values generated)
```

This looks better – we could have done the same thing by replacing *x3* with a blank string if *bmi* was missing, but it's better to create the variable correctly in the first place.

Now, we don't actually need any of these variables, so they can be dropped without a problem:

```
drop x* units
```

4.5 Manipulating Dates

Dates in Stata are a little odd if you've never used dates in a statistical program or Excel before. A date in Stata is coded as a number, specifically the number of days since the 1st of January 1960: for example, the 1st of January 2020 in Stata is coded as 21,915. These numbers are then formatted to look like dates. However, when using *if* statements that include a date, it's important to remember that dates are numbers.

There's a couple of helpful functions it's worth knowing about to manipulate dates. The first is the **`date()`** function, which converts a string that looks (to us) like a date into a number Stata can interpret as a date. The general syntax is:

```
date({string variable or date},"{a combination of D, M and Y}",{maximum year})
```

The first part of the function is a string variable (or string) containing something that looks (to us) like a date. The second part of the function is a combination of "D", "M" and "Y", representing days, months and years respectively. This part tells Stata what to expect the order of the date to look like: Stata can handle both numeric (1 = January, 2 = February etc.) and string months ("Jan" = January, "February" = February etc.), but it needs to know how the days, months and years are ordered.

The third part of the function tells Stata what the maximum possible year is. This is only relevant if only the last two digits of the year are given, for example 01/01/20 to mean the 1st of January 2020. If you give Stata a maximum possible year, it will fill in the first two digits of the year with the century that is closest to the maximum possible year without going over. For example, if the year is given as 01/01/20 and the maximum possible year as 2050, then Stata would interpret the date as the 1st of January 2020. If the maximum possible year was 1950, then Stata would interpret the date as the 1st of January 1920.

As with many function, **`date()`** can be used with variables or strings. If you want to know the number Stata will store a date as, for example 01/01/2020, you can type:

```
display date("01/01/2020","DMY")
21915
```

This is how we know that Stata interprets 01/01/2020 as 21,915. Just to be absolutely sure, we can make Stata format a displayed number as a date by putting the formatting code after **`display`**:

```
display %td 21915
01jan2020
```

This looks good. Let's do a couple more examples:

```
display date("03-14-2013","MDY")
19431
display %td date("03-14-2013","MDY")
14mar2013
```

Here, the date is in the usual American format, month-day-year, so we've rearranged "DMY" to "MDY". We've also included the **%td** format, so the second command is displayed as a date so we can make sure Stata is interpreting the date correctly.

```
display date("01 Jan 20","DMY",2000)
-14610
display %td date("01 Jan 20","DMY",2000)
01jan1920
```

Here, we've used words for the month and set the maximum year as the year 2000, so Stata interprets 01/01/20 as 01/01/1920. Stata can handle all kinds of date formats, and negative dates are just dates before the 1st of January 1960.

Ok, let's put this function to use now by turning *date_of_bp_measurement* into a numeric variable and format it to look like a date:

```
gen date_of_bp_measurement_2 =
date(date_of_bp_measurement,"DMY")
(109 missing values generated)
format date_of_bp_measurement_2 %td
label variable date_of_bp_measurement_2 "Date of blood
pressure measurement"
list date_of_bp_measurement date_of_bp_measurement_2 in 1/5
```

	date~ement	date_of~2
1.	03-12-2018	03dec2018
2.	31-12-2018	31dec2018
3.	24-09-2018	24sep2018
4.	21-07-2017	21jul2017
5.	24-08-2019	24aug2019

Great, this looks right.

Depending on what we're doing, we may actually want to convert a numeric date into a string. This can be done with the **string()** function we used last lesson, as the function can be given a format after the variable (or string):

```
gen x = string(date_of_bp_measurement_2,"%td")
list date_of_bp_measurement_2 x in 1/5
```

	date_of~2	x
1.	03dec2018	03dec2018
2.	31dec2018	31dec2018
3.	24sep2018	24sep2018
4.	21jul2017	21jul2017
5.	24aug2019	24aug2019

```
drop x
```

The final useful set of functions we'll use with dates allow us to extract the day, month or year from a numeric date, for example:

```
gen day = day(date_of_bp_measurement_2)
(109 missing values generated)
list date_of_bp_measurement_2 day in 1/5
```

	date_of~2	day
1.	03dec2018	3
2.	31dec2018	31
3.	24sep2018	24
4.	21jul2017	21
5.	24aug2019	24

```
gen month = month(date_of_bp_measurement_2)
(109 missing values generated)
list date_of_bp_measurement_2 month in 1/5
```

	date_of~2	month
1.	03dec2018	12
2.	31dec2018	12
3.	24sep2018	9
4.	21jul2017	7
5.	24aug2019	8

```
gen year = year(date_of_bp_measurement_2)
(109 missing values generated)
list date_of_bp_measurement_2 year in 1/5
```

	date_of~2	year
1.	03dec2018	2018
2.	31dec2018	2018
3.	24sep2018	2018
4.	21jul2017	2017
5.	24aug2019	2019

```
drop day month year
```

That's it for dates. Let's remove the string version of *date_of_bp_measurement* and rename *date_of_bp_measurement_2*:

```
drop date_of_bp_measurement
rename date_of_bp_measurement_2 date_of_bp_measurement
```

Ok, let's save the dataset with our initials and the lesson number, and we'll move on to the exercise:

```
save "Lesson_04_SH.dta", replace
```

4.6 Exercise

For this exercise, see how you do with the following:

1. Change your directory (if necessary) and load in the **Exercise_03_{initials}.dta** dataset from the first exercise
    a. If you're not confident with your dataset, you can load in the **Exercise_04.dta** dataset instead

2. Convert *calories* to a string variable called *calories_string*, with 1 decimal place, then convert it back to a numeric variable called *calories_numeric*
    a. Check that your commands worked as expected
    b. Will the newly created numeric variable be exactly the same as the original *calories* variable?
    c. Now remove the new variables

3. Encode *sex* using the already created label *sex*, creating a new variable *sex3*, and order it to be next to *sex2*
    a. Check that all the *sex* variables have the same information
    b. If they do, keep *sex2* or *sex3*, drop the others and rename the kept variable to *sex*

4. Merge in the **Exercise_04_merge.dta** dataset using 1:1 merging with the *id* variable
    a. Either remove the *_merge variable*, or use merge with the nogenerate option

5. Find and fix the spelling error in *favourite_dog_breed* using the subinstr() function

6. Convert the *favourite_dog_breed* variable into 4 separate variables called *breed_1* to *breed_4*, using the semicolons as delimiters
    a. Label each variable and order the 4 separate variables after the *favourite_dog_breed* variable
    b. Recreate the *favourite_dog_breed* variable using the 4 separate variables and the concat() function of egen or a straight generate command, then check the recreated variable is identical to the *favourite_dog_breed* variable
    c. If it is, drop the *favourite_dog_breed* and the concatenated variable(s)
    d. Remove all leading and trailing spaces from the new breed variables

7. There are 9 types of dog breed: check whether each breed variable includes all breeds
    a. Use the length() function to find out the character length the longest dog breed
    b. Remove any temporary variables you may have created

8. Turn all the breed variables into labelled numeric variables: make sure they all have the same value label, and give them variable labels too
    a. Check that all the string and labelled numeric variables match, then remove all the string breed variables and rename the labelled numeric breed variables the same as the original string breed variables

9. Convert *date_of_bp_measurement* to a formatted numeric date variable, including dropping the original string variable, and renaming, labelling and ordering the newly created numeric variable after *date_of_recruitment*
    a. Make sure the dates are correct before dropping the original variable though

10. Format the *calories* variable to not show any decimal places, *weight* to show 2 decimal places, and *bmi* to show 1 decimal place
    a. Also format *date_of_recruitment* into a date variable

11. Save the modified dataset, calling it **Exercise_04_{initials}.dta**

4.7 Exercise – Answers

1. As before:
   a. `use "Exercise_03_SH.dta", clear`
   b. Or, if you're not confident in your save from exercise 3:
   c. `use "Exercise_04.dta", clear`

2. You can use either the **`tostring`** command or **`string()`** function, and the **`destring`** command or **`real()`** function:
   a. `gen calories_string = string(calories,"%9.1f")`
   b. `*OR*`
   c. `tostring calories, gen(calories_string) format(%9.1f) force`
   d. Then:
   e. `gen calories_numeric = real(calories_string)`
   f. `*OR*`
   g. `destring calories_string, gen(calories_numeric)`
   h. The newly created numeric variable will necessarily be limited to 1 decimal place, since that's how many decimal places the string was created with. Therefore, the new variable won't be exactly the same as the original *calories* variable, as that has more than 1 decimal place. It will be the same to 1 decimal place though.
   i. `drop calories_*`

3. **`endode`** command:
   a. `encode sex, gen(sex3) label(sex)`
   b. `order sex3, a(sex2)`
   c. `tab sex sex3, miss`
   d. `tab sex2 sex3, miss`
   e. `drop sex sex2`
   f. `rename sex3 sex`

4. **`merge`** command:
   a. `merge 1:1 id using "Exercise_04_merge.dta", nogen`

5. **`replace`** command:
   a. `replace favourite_dog_breed = subinstr(favourite_dog_breed,"Retriver","Retriever",.)`

6. **split**, **label**, **order**, **generate** and **egen** commands:
   a. `split favourite_dog_breed, gen(breed_) parse(;)`
   b. `label variable breed_1 "1st favourite dog breed"`
   c. `label variable breed_2 "2nd favourite dog breed"`
   d. `label variable breed_3 "3rd favourite dog breed"`
   e. `label variable breed_4 "4th favourite dog breed"`
   f. `order breed_*, a(favourite_dog_breed)`
   g. `egen favourite_dog_breed_2 = concat(breed_1 breed_2 breed_3 breed_4), punct(;)`
   h. `count if favourite_dog_breed == favourite_dog_breed_2`
   i. `*OR*`
   j. `gen favourite_dog_breed_3 = breed_1 + ";" + breed_2 + ";" + breed_3 + ";" + breed_4`
   k. `count if favourite_dog_breed == favourite_dog_breed_3`
   l. `drop favourite*`
   m. `replace breed_1 = strtrim(breed_1)`
   n. `replace breed_2 = strtrim(breed_2)`
   o. `replace breed_3 = strtrim(breed_3)`
   p. `replace breed_4 = strtrim(breed_4)`

7. **tab** and **gen** commands mainly:
   a. `tab breed_1`
   b. `tab breed_2`
   c. `tab breed_3`
   d. `tab breed_4`
      i. We can see by tabulating the breed variables that each variable has 9 breeds, and they all look the same in each variable
   e. `gen x = length(breed_1)`
   f. `sum x, detail`
   g. `*OR*`
   h. `tab breed_1 x`
      i. Either by summarizing or tabulating *breed_1* and *x*, we can see the "German Shorthaired Pointer" has the most characters, at 26 characters
   i. `drop x`

8. **encode, tabulate** and **rename** commands:
    a. `encode breed_1, gen(breed_1x) label(breed)`
    b. `encode breed_2, gen(breed_2x) label(breed)`
    c. `encode breed_3, gen(breed_3x) label(breed)`
    d. `encode breed_4, gen(breed_4x) label(breed)`
    e. `tab breed_1 breed_1x`
    f. `tab breed_2 breed_2x`
    g. `tab breed_3 breed_3x`
    h. `tab breed_4 breed_4x`
        i. All the tabulations look the same, so the encoding worked
    i. `drop breed_1-breed_4`
    j. `rename breed_1x breed_1`
    k. `rename breed_2x breed_2`
    l. `rename breed_3x breed_3`
    m. `rename breed_4x breed_4`

9. **generate, format, label** and **order** commands:
    a. `gen date_of_bp_measurement_2 =`
       `date(date_of_bp_measurement,"DMY")`
    b. `format date_of_bp_measurement_2 %td`
    c. `label variable date_of_bp_measurement_2 "Date of`
       `blood pressure measurement"`
    d. `order date_of_bp_measurement_2,`
       `a(date_of_recruitment)`
    e. `drop date_of_bp_measurement`
    f. `rename date_of_bp_measurement_2`
       `date_of_bp_measurement`

10. **format** commands:
    a. `format calories %9.0f`
    b. `format weight %9.2f`
    c. `format bmi %9.1f`
    d. `format date_of_recruitment %td`

11. And a **save** command:
    a. `save "Exercise_04_SH.dta", replace`