Stat 342 - Wk 4

SQL

'inner join'

Data Step

Making new variables from old ones

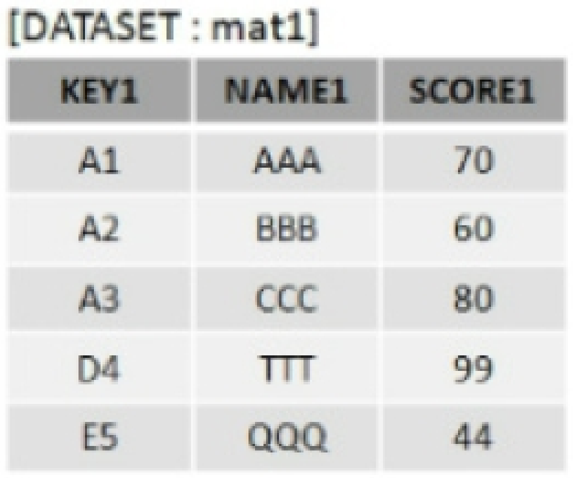
The basics of missing values

Inner Joins!

Inner joins are the most popular of several ways to combine two or more datasets. If someone refers simply to a 'join', it is usually an inner join.

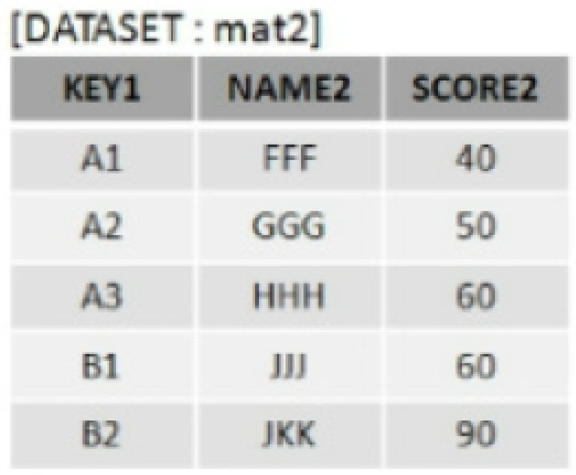
A select statement with an inner join takes variables from both datasets and matches them up according to some variable in comment, such as userID or date.

Let 'Key1' be the joining variable



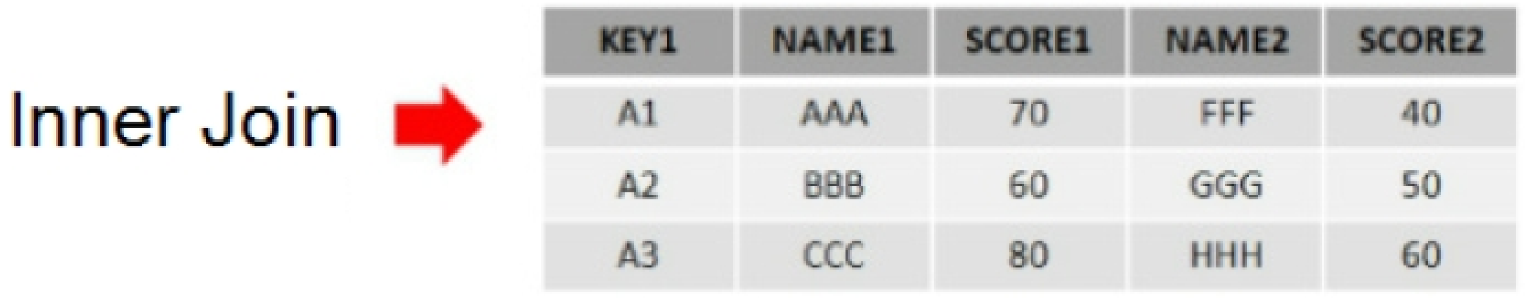
[DATASET : mat1]

KEY1	NAME1	SCORE1
A1	AAA	70
A2	BBB	60
A3	CCC	80
D4	TTT	99
E5	QQQ	44



[DATASET : mat2]

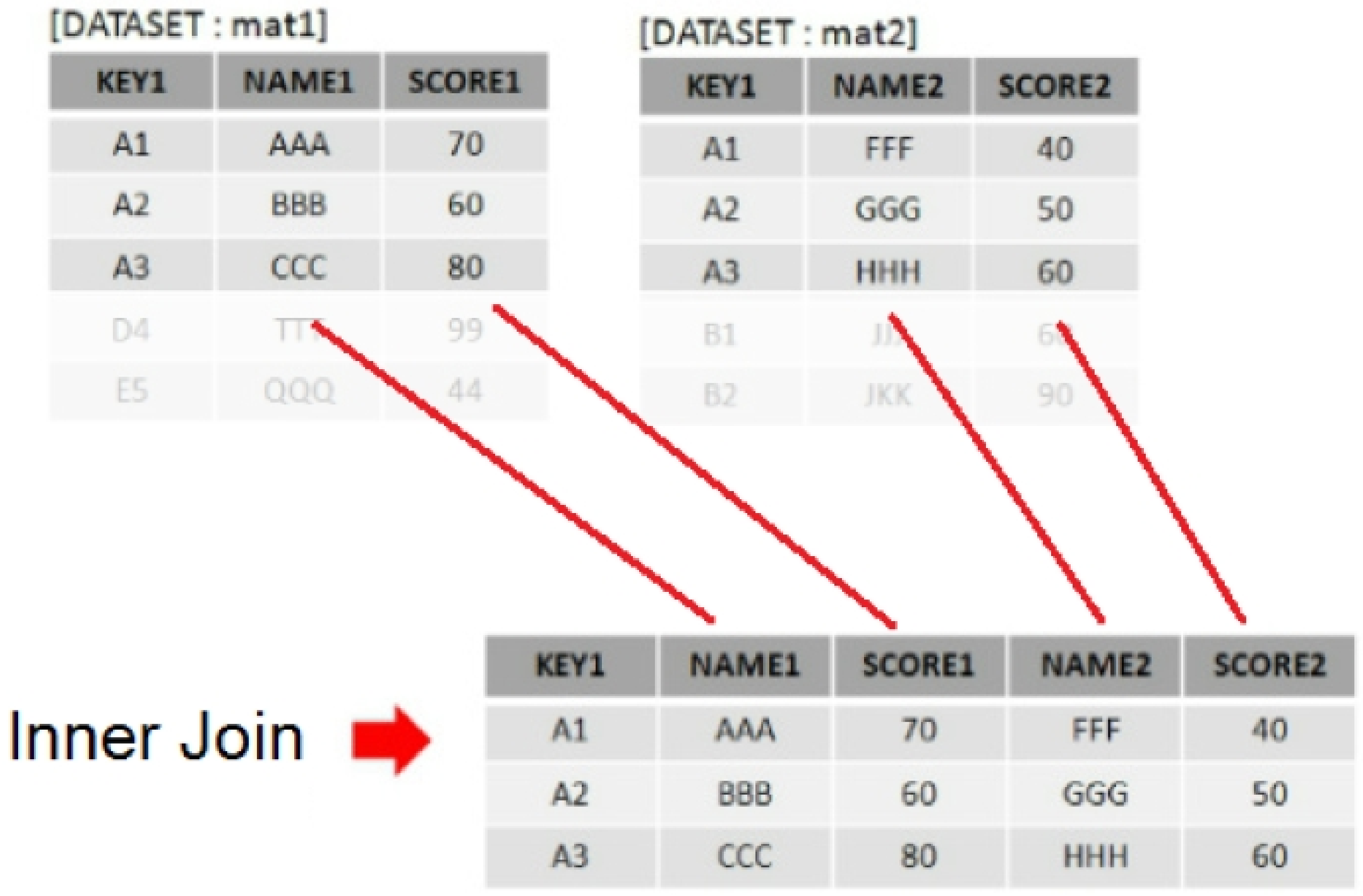
KEY1	NAME2	SCORE2
A1	FFF	40
A2	GGG	50
A3	HHH	60
B1	JJJ	60
B2	JKK	90



Inner Join

KEY1	NAME1	SCORE1	NAME2	SCORE2
A1	AAA	70	FFF	40
A2	BBB	60	GGG	50
A3	CCC	80	HHH	60

An inner join makes a new dataset with one row for each matched variable value and the chosen variables from each.



[DATASET : mat1]

KEY1	NAME1	SCORE1
A1	AAA	70
A2	BBB	60
A3	CCC	80
D4	TTT	99
E5	QQQ	44

[DATASET : mat2]

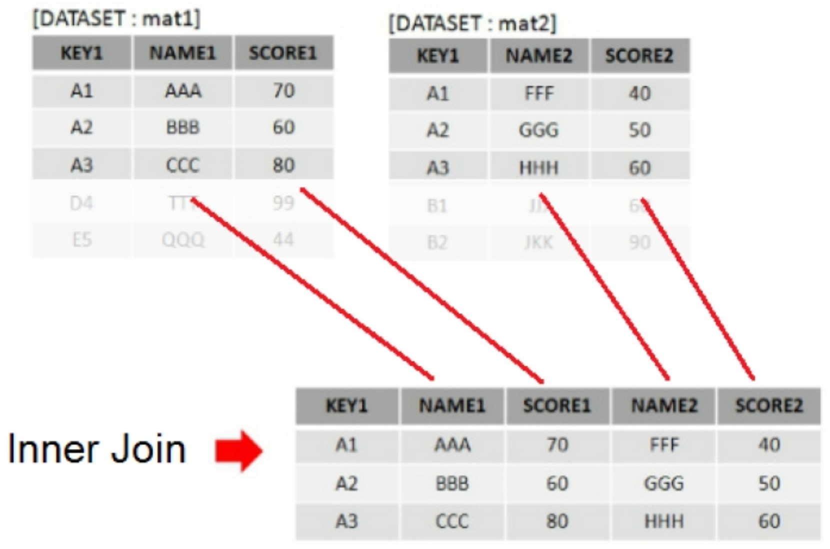
KEY1	NAME2	SCORE2
A1	FFF	40
A2	GGG	50
A3	HHH	60
B1	JJJ	6[illegible]
B2	JKK	90

Inner Join

KEY1	NAME1	SCORE1	NAME2	SCORE2
A1	AAA	70	FFF	40
A2	BBB	60	GGG	50
A3	CCC	80	HHH	60

The code to do this would be

```
select *
from mat1
inner join mat2
on mat1.key1 = mat2.key1;
```



[DATASET : mat1]

KEY1	NAME1	SCORE1
A1	AAA	70
A2	BBB	60
A3	CCC	80
D4	TT	99
E5	QQQ	44

[DATASET : mat2]

KEY1	NAME2	SCORE2
A1	FFF	40
A2	GGG	50
A3	HHH	60
B1	JJ	6
B2	JKK	90

Inner Join

KEY1	NAME1	SCORE1	NAME2	SCORE2
A1	AAA	70	FFF	40
A2	BBB	60	GGG	50
A3	CCC	80	HHH	60

In this code, the Y2000 variable from each of two different data sets, teen fertility and school years. The joining variable found in both datasets is 'country'.

```
proc sql;
   select wk03teenfertility.Country,
   wk03teenfertility.Y2000 as fertility2000,
   wk03schoolyears.Y2000 as school2000

   from wk03teenfertility
   inner join wk03schoolyears
   on wk03teenfertility.Country =
wk03schoolyears.Country;
```

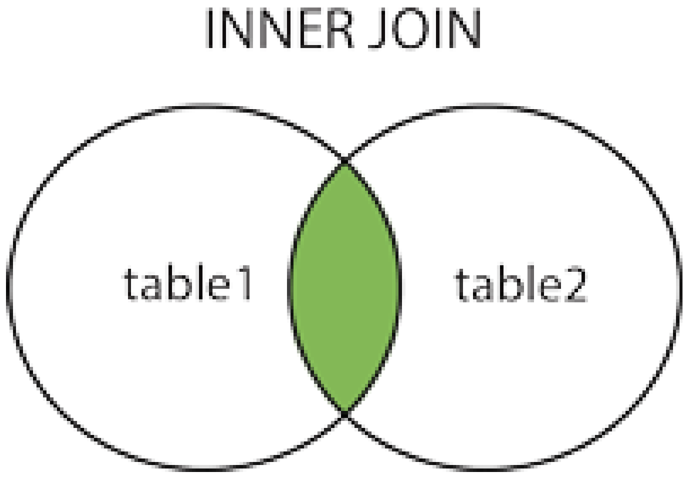
A select statement with an inner join has syntax like this:

```
select <dataset 1>.<var>,...<dataset 2>.<var>
   from <dataset 1>
   inner join <dataset 2>
   on <index variable in both datasets>;
```

The most common condition used here is dataset1.variable = dataset2.variable

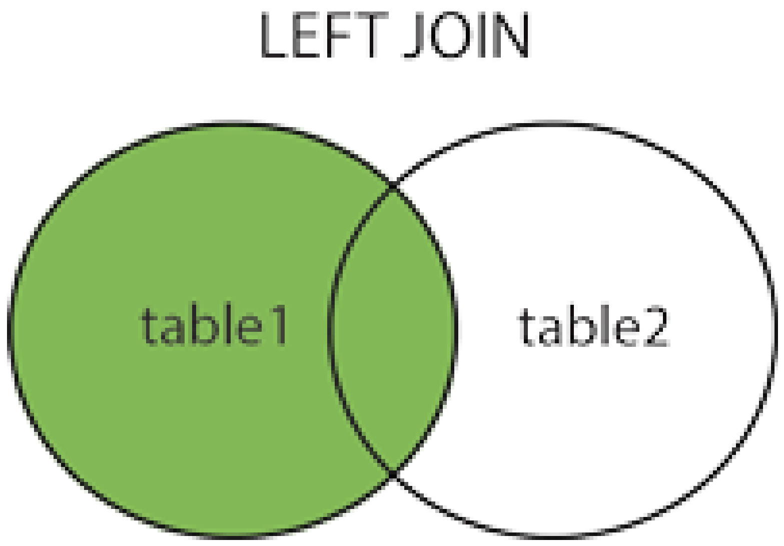
With an inner join, only 1 row is created for each instance where a value from the first join variable ***matches*** a value from second join.

In other words, for a row to be in an inner join, its index value has to be in BOTH of the tables.



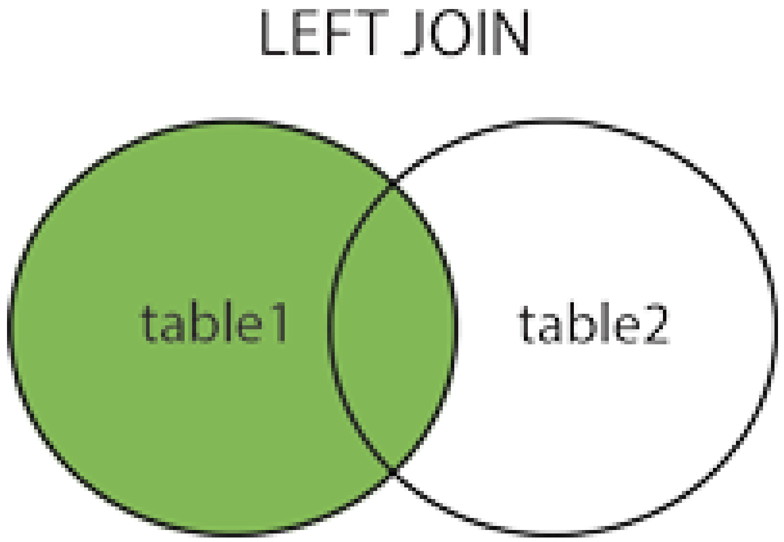
There are other kinds of joins, such as left join, right join, and outer join.

The result of a left join is similar to that of an inner join: A table of the selected rows from both tables and the values in common when they match, just like an inner join.



However, unlike an inner join...

For a row to be in a left join, its index value only has to be in the first table.



The other variables in the table that results from a left join, will be filled for any index values from both tables, but for the index values only in table 1, there will be missing data.

Compare the 'try it yourself' demonstrations at

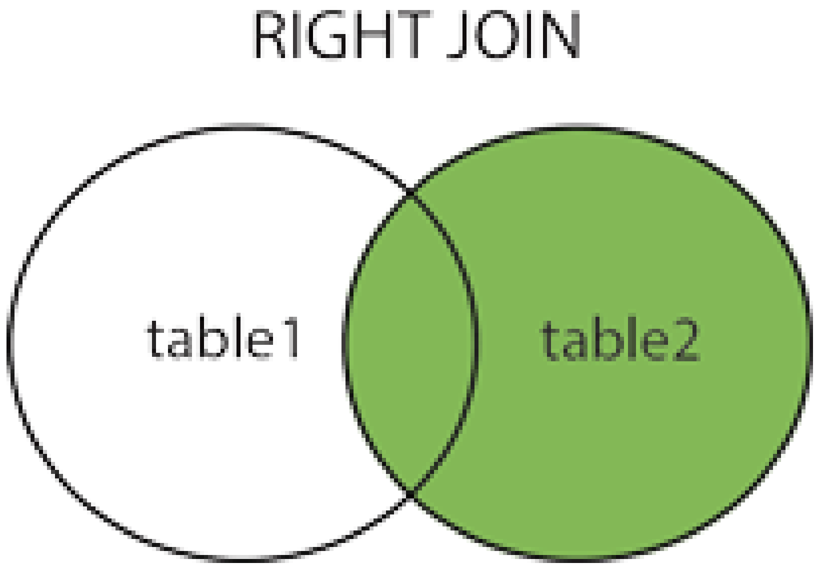
http://www.w3schools.com/sql/sql_join_inner.asp

and at

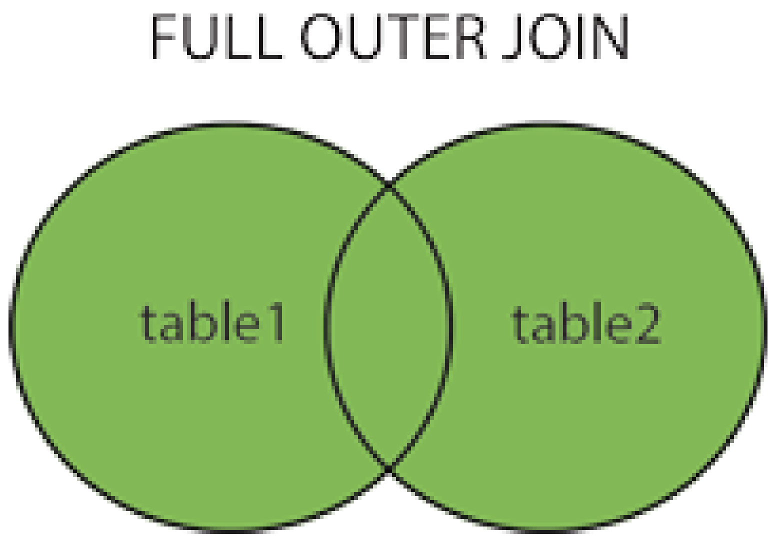
http://www.w3schools.com/sql/sql_join_left.asp

Likewise, a right join includes rows for any index values that are found in table2.

The only real difference between a left join and a right join is the order of the tables.



A 'Full join', or 'outer join' includes rows for index values that are in EITHER TABLE.



Country dataset examples on paper.

Data step: Making new variables from old ones

In the data step, there is a lot of flexibility with what we can do with variables.

This includes making derived variables.

Derived variables are useful as summaries or combinations of several variables, or as rescaling of variables.

They also allow you to make a copy of a variable that you can change without affecting the original data.

Since there are a lot more possibilities to making derived variables, the basic syntax is more vague.

```
data <outputname>;
   set <inputname>;
   <newvar> = <function of oldvars>;
run;
```

You can take an annual income and convert it into a weekly income by dividing it by 52.

```
data finance;
   set finance;
   w_income = year_income / 52;
run;
```

In this week's lab, we use a variable named 'fertility', which is the number of births in a year per 1000 teenage girls.

To rescale that into the number of children in a lifetime (30 fertile years), or the Total Fertility Rate, we make a new variable TFR, and define it as a function of fertility.

```
data wk04part4;
   set wk04part3;
   TFR = fertility / 1000 * 30;
run;
```

Also, we can remove the original variable if we wish. The drop command shown here is applied at the end of processing. It deletes the variable from the final dataset, but only after we have a chance to use it in the data step.

```
data wk04part4;
   drop fertility;
   set wk04part3;
   TFR = fertility / 1000 * 30;
run;
```

We can make variables based on IF-THEN conditions as well.

Code that determines what happens based on certain conditions, (if, then, else, do, while) are called control-of-flow commands.

```
data finance2;
   set finance1;
   if year_income > 5000 then over5k = 1;
   if year_income < 5000 then over5k = 0;
run;
```

A note on variable declarations.

In some other programming languages, you have you *declare* a variable before you use it. That means telling the computer the name of the variable and its format.

You can directly decide the format of a SAS variable (covered later in this semester if time permits) before using it.

However, you can also just start using it without extra work, and SAS will do its best to determine the format.

In this example, assume that the variable over5k is not part of the dataset 'finance1', so this is the first time that SAS has seen it.

The first value assigned to it is a number, so the default numeric format is used.

```
data finance2;
   set finance1;
   if year_income > 5000 then over5k = 1;
   if year_income <= 5000 then over5k = 0;
run;
```

You could also replace these two IF statements with a single IF-ELSE statement. This works the same way, but is considered more protected against mistakes because there's no way to accidentally run both IF statements on the same line.

```
data finance2;
   set finance1;
   if year_income > 5000 then over5k = 1;
   else over5k = 0;
```

```
run;
```

The data step runs once for every row in the dataset mentioned in 'set'. (If there are multiple datasets listed, it runs once for every row in the first one, then the second one, and so on)

Once a new variable is defined, SAS will make room for it in every row.

If a value for that variable can't be found, then it's left as a ***missing value***.

So what would happen in this case when speed is between 10 and 20?

```
data finance2;
   set finance1;
   if speed > 20 then fast = 1;
   if speed < 10 then fast = 0;
run;
```

If the original variable being used to make a new variable has a missing value, the new variable may result in a missing value too.

In the annual/weekly income example, an unknown value divided by 52 is still unknown. So the variable w_income will be missing for any of those cases.

In the speed/fast example, assume the value for speed is missing. By default, a missing value is treated as the lowest possible value, so 'fast' will be 0 for missing speed.

1) On-paper example of summary statistics and missing data.

2) On-paper example of IF statements and missing data.

Let's say we didn't want missing speed values to be summarized as fast = 0. Instead, we wanted to be more accurate and say that 'speed' was missing, then 'fast' should be to.

We can fix this by making two changes.

1) Including an AND in the second if statement to make it need two conditions together.

2) Make that second condition 'not missing'.

The value for missing is just a period .

We can exclude the missing value cases with either 'fast > .' , or with 'fast ne . '

```
data finance2;
   set finance1;
   if speed > 20 then fast = 1;
   if (speed < 10) and (speed ne .)
      then fast = 0;
run;
```

We may also want to include an explicit statement declaring 'fast' as missing before we look at speed. This doesn't change the result, but it makes the code more readable to a human.

```
data finance2;
   set finance1;
   fast = .;
   if speed > 20 then fast = 1;
   if (speed < 10) and (speed ne .)
      then fast = 0;
run;
```

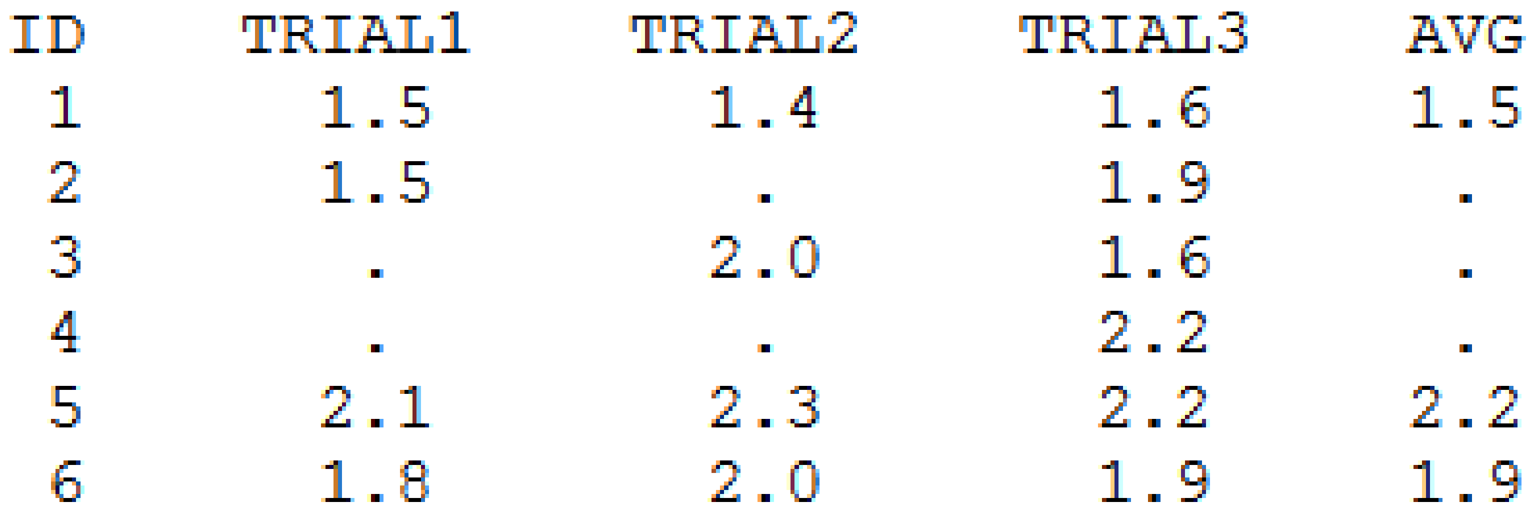
Variables can be made from several other variables at once.

Here we have a dataset times, and we compute average time from three different trials.

```
DATA times2 ;
  SET times ;
  avg = (trial1 + trial2 + trial3) / 3 ;
RUN ;
```

If ANY of the three trail times are NA (missing), avg will be.

SAS can't look at an arbitrary, user-made formula and know which of the variables are important or what to do if they are missing. Therefore, if any are missing, it doesn't try to calculate anything for that variable for that row.



```
ID     TRIAL1     TRIAL2     TRIAL3     AVG
 1      1.5        1.4        1.6       1.5
 2      1.5         .         1.9        .
 3       .         2.0        1.6        .
 4       .          .         2.2        .
 5      2.1        2.3        2.2       2.2
 6      1.8        2.0        1.9       1.9
```

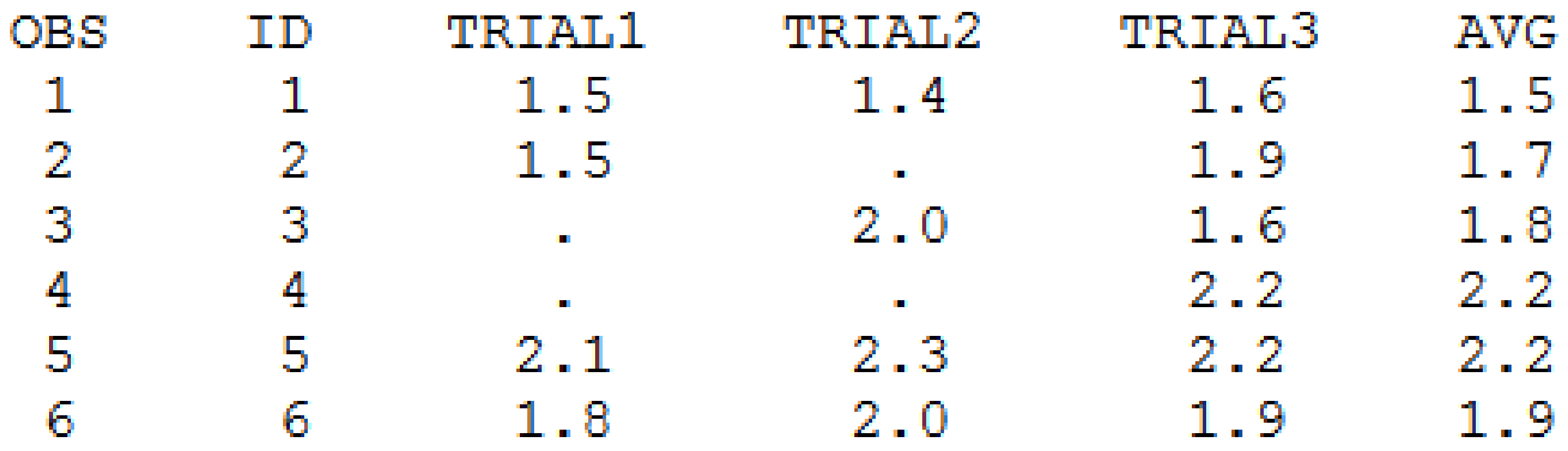
We can also use basic functions to compute variables.

```
DATA times2 ;
  SET times ;
  avg = MEAN(trial1, trial2, trial3);
  sd = SD(trial1, trial2, trial3);
  Ntrials = N(trial1, trial2, trial3);
RUN ;
```

If ALL of the three trail times are NA (missing), avg will be.

But if only some trials are missing, an average is taken from the trial times that remain.

Using MEAN(), we get a value for avg in each row.



```
OBS    ID    TRIAL1    TRIAL2    TRIAL3    AVG
 1      1     1.5       1.4       1.6      1.5
 2      2     1.5        .        1.9      1.7
 3      3      .        2.0       1.6      1.8
 4      4      .         .        2.2      2.2
 5      5     2.1       2.3       2.2      2.2
 6      6     1.8       2.0       1.9      1.9
```

The function MEAN() is smart enough to know that you can take an average from the remaining values if some are missing.

Other functions of single variables include...

INT(x), which rounds x down to the nearest integer.

ROUND(x). which rounds x up or down to the nearest whole.

ROUND(x, .3), which rounds x to the nearest .001. (the .3 means 3 digits after the decimal point).

SQRT(x), LOG(x), EXP(x), which do the square root, natural log, and natural exponent, respectively.

And other functions of several variables together include...

MEDIAN(OF x1-x119)

NMISS(OF x1-x22)   for the number of missing values

SUM(OF x3-x5) for the sum (ignoring missing)

Recall that you can refer to a whole range of variables with hyphen notation like x1 – x20 for x1, x2, ... , x19, x20.

We need to include 'OF' in those cases to make sure SAS doesn't think it's a subtraction between two variables.

`Additional references for these topics.`

`http://www.ats.ucla.edu/stat/sas/modules/funct.htm`

`http://www.ats.ucla.edu/stat/sas/modules/missing.htm`

`http://www.ats.ucla.edu/stat/sas/modules/vars.htm`

The RETAIN command

In a default data step, every row is independent of every other row. What you calculate in one row doesn't affect future rows.

However, there are times when you would rather have things carry over.

Case 1: Assume each row in personal_finance represents one week's budget. We can find year-to-date (YTD) income. This is useful for seeing how much income you've earned since the beginning of the year, and for finding the total at the end of the year.

```
DATA personal_finance2;
  SET personal_finance;
  YTD = YTD + week_income;
  retain YTD = 0;
RUN ;
```

Case 2: If each row of a dataset is the amount of sales for that year, you may want to compare information to the previous year.

Here, YoY stands for year-over-year growth. sales_lastyear is set to the value of sales_thisyear after YoY is calculated.

```
DATA corp_finance2;
  SET corp_finance;
  YoY = sales_thisyear / sales_lastyear - 1;
   sales_lastyear = sales_thisyear;
  retain lastyear;
RUN ;
```

On paper-example of case 1 of the retain command.

On paper-example of case 2 of the retain command.

Random number generation

The most common functions to generate random numbers from are the uniform and the normal distribution.

This is done with the RAND() function inside a data step, specifying a distribution, and parameters if necessary.

RAND('UNIFORM') will provide a random value from 0 to 1.

RAND('NORMAL') will provide a random value from the standard normal distribution (mean = 0, sd = 1).

Why use random number generation?

1) **Sampling**. If you had a large dataset of every cellphone number in Vancouver, and you wanted to get the opinion of 1000 randomly selected people. That random selection is done with random number generation.

You may want to...

...weight your sample to account for certain demographics not answering their phones.

...give the possible responses to a multiple choice question in a randomly selected order.

Why use random number generation?

2) Goodness of fit testing.

If you wanted to find out how a certain set of data would behave if it followed a hypothesized distribution, you could generate values from that distribution and explore that hypothetical situation.

You could see how good that distribution fits your data by comparing hypothetical data to real data. That's one way to assess goodness of fit.

Why use random number generation?

3) Making data anonymous. (1/2)

If you are going to be sharing a dataset with other researchers or the public, you have an obligation to protect the privacy of any people whose data is recorded.

Sometimes private data like phone numbers or e-mail addresses is used to identify people in a data set. For example, in a record of sales, where one row is one sale, you might see the same phone number in multiple rows.

3) Making data anonymous. (2/2)

If that's the case, you would be destroying useful information by getting rid of the phone number variable.

What you can do, however, is scramble the phone numbers. They would need to be scrambled in such a way that the same number gets scrambled the same way every time.

That way, someone else could read the data after it has been scrambled and still see when one person has made many purchases. They cannot, however, call that person.

Random Number Generation: Seeds

Computers cannot (typically) generate true random numbers. Instead, they use a complicated formula based on a starting value that has to be provided by an outside source.

When you use a random number function like UNIFORM(x),

The value x is the starting value, or seed, that is used.

In SAS, by default

the computer will use the time of its internal clock as its seed.

If you specify a positive integer like 345 in the streaminit() routine with `call streaminit(345)`

Then that value '345' will be used as the first seed. When a random number is generated, a new seed based on '345' will be used.

Why care about the seed?

If the clock-based seed is used, there is no way to retrieve a seed and use it again. Every time you run an analysis on a time-based seed you will get a different result.

If you want to generate random numbers, but you want to generate the same random numbers every time you run an analysis that includes setting a fixed seed, it will give the same result every time.

Here is an example program that sets a fixed seed and generates 10 random numbers from the Cauchy distrubtion.

```
data random;
   call streaminit(123);
   do i=1 to 10;
      x1=rand('cauchy');
      output;
   end;
run;
```

The same 10 Cauchy values will be found every time.

SAS can generate random numbers from a wide variety of distributions and parameters sets.

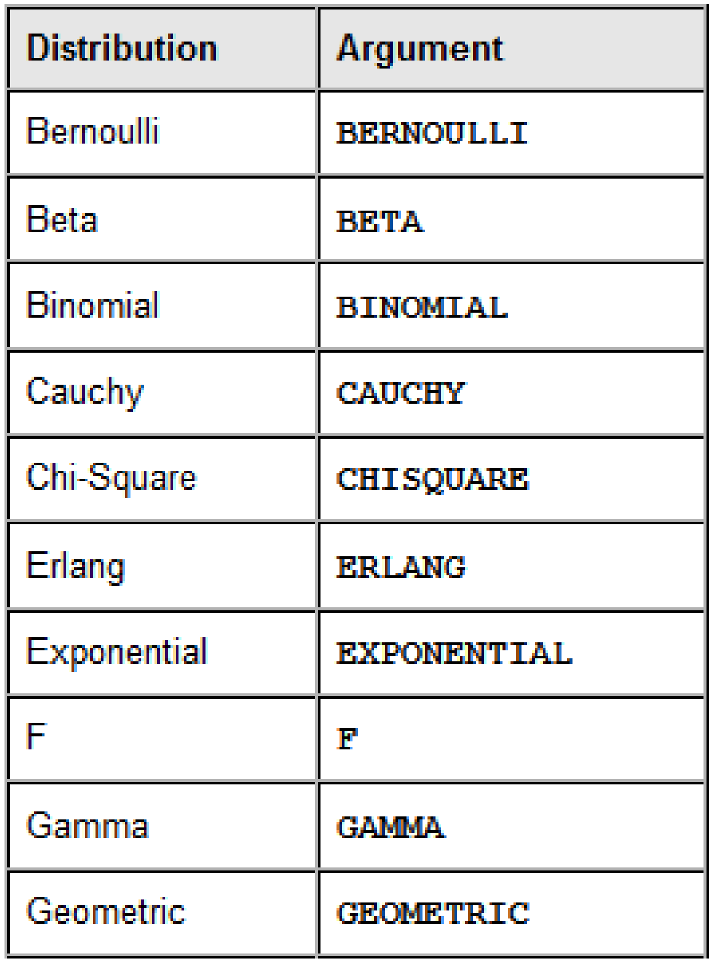
RAND('NORMAL', 5,3)

will give you a random normal (aka Gaussian) number from a distribution with mean 5 and standard deviation 3.

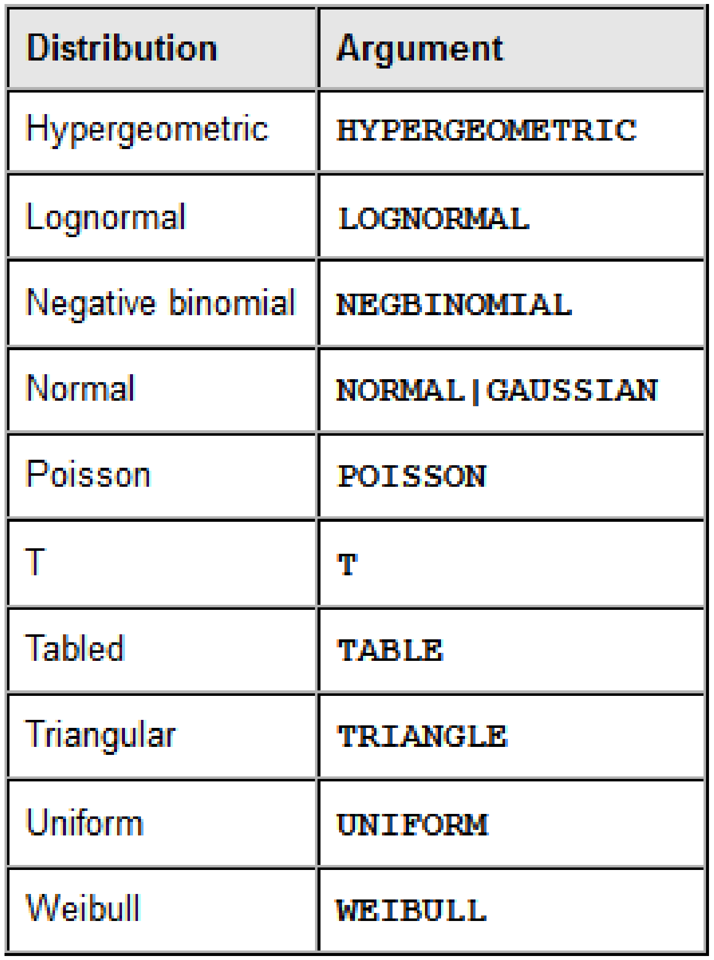
RAND('POISSON', 10)

will provide a random number from a Poisson distribution with lambda (mean, variance) of 10.

From SAS Documentation on the RAND function



Distribution	Argument
Bernoulli	`BERNOULLI`
Beta	`BETA`
Binomial	`BINOMIAL`
Cauchy	`CAUCHY`
Chi-Square	`CHISQUARE`
Erlang	`ERLANG`
Exponential	`EXPONENTIAL`
F	`F`
Gamma	`GAMMA`
Geometric	`GEOMETRIC`



Distribution	Argument
Hypergeometric	`HYPERGEOMETRIC`
Lognormal	`LOGNORMAL`
Negative binomial	`NEGBINOMIAL`
Normal	`NORMAL|GAUSSIAN`
Poisson	`POISSON`
T	`T`
Tabled	`TABLE`
Triangular	`TRIANGLE`
Uniform	`UNIFORM`
Weibull	`WEIBULL`

On-paper example of random number generation.

(Reference slides follow)

Special Variables

There are a few variables that are present in every SAS data step that you can use. These are typically for debugging data steps.

_n_ , which tracks the number of iterations (rows) that the data step has gone through already.

error , which is 1 if there was an error processing a row, and 0 otherwise.

Labels example.

```
DATA  auto2;
   SET auto;
   LABEL  rep78  ="1978 Repair Record"
          mpg    ="Miles Per Gallon"
          foreign="Where Car Was Made";
RUN;

PROC CONTENTS DATA=auto2;
RUN;
```