Anyone Can Learn PROC TABULATE, v2.0

Lauren Haworth
Ischemia Research & Education Foundation
San Francisco

➢ ABSTRACT

SAS® Software provides hundreds of ways you can analyze your data. You can use the DATA step to slice and dice your data, and there are dozens of procedures that will process your data and produce all kinds of statistics. But odds are that no matter how you organize and analyze your data, you'll end up producing a report in the form of a table.

This is why every SAS user needs to know how to use PROC TABULATE. While TABULATE doesn't do anything that you can't do with other PROCs, the payoff is in the output. TABULATE computes a variety of statistics, and it neatly packages the results in a single table.

Unfortunately, TABULATE has gotten a bad rap as being a difficult procedure to learn. This paper will prove that if you take things step by step, anyone can learn PROC TABULATE.

➢ INTRODUCTION

This paper will start out with the most basic one-dimensional table. We will then go on to two-dimensional tables, tables with totals, and finally, three-dimensional tables. By the end of this paper, you will be ready to build most basic TABULATE tables.

➢ ONE-DIMENSIONAL TABLES

To really understand TABULATE, you have to start very simply. The simplest possible table in TABULATE has to have three things: a PROC TABULATE statement, a TABLE statement, and a CLASS or VAR statement. In this example, we will use a VAR statement. Later examples will show the CLASS statement.

The PROC TABULATE statement looks like this:

```
PROC TABULATE DATA=TEMP;
```

The second part of the procedure is the TABLE statement. It describes which variables to use and how to arrange the variables. This first table will have only one variable, so you don't have to tell TABULATE where to put it. All you have to do is list it in the TABLE statement. When there is only one variable, you get a one-dimensional table.

```
PROC TABULATE DATA=TEMP;
  TABLE RENT;
RUN;
```

If you run this code as is, you will get an error message because TABULATE can't figure out whether the variable RENT is intended as an analysis variable, which is used to compute statistics, or a classification variable, which is used to define row or column categories in the table.

In this case, we want to use rent as the analysis variable. We will be using it to compute a statistic. To tell TABULATE that RENT is an analysis variable, you use a VAR statement. The syntax of a VAR statement is simple: you just list the variables that will be used for analysis. So now the syntax for our PROC TABULATE is:

```
PROC TABULATE DATA=TEMP;
  VAR RENT;
  TABLE RENT;
RUN;
```

The result is the table shown below. It has a single column, with the header RENT to identify the variable, and the header SUM to identify the statistic. There is just a single table cell, which contains the value for the sum of RENT for all of the observations in the dataset TEMP.

```
„ƒƒƒƒƒƒƒƒƒƒƒƒ†
,     RENT     ,
‡ƒƒƒƒƒƒƒƒƒƒƒƒ‰
,     SUM      ,
‡ƒƒƒƒƒƒƒƒƒƒƒƒ‰
,   134582.00  ,
ŠƒƒƒƒƒƒƒƒƒƒƒƒŒ
```

➢ ADDING A STATISTIC

The previous table shows what happens if you don't tell TABULATE which statistic to use. If the variable in your table is an analysis variable, meaning that it is listed in a VAR statement, then the statistic you will get by default is the sum. Sometimes the sum will be the statistic that you want. Most likely, sum isn't the statistic that you want.

To add a statistic to a PROC TABULATE table, you modify the TABLE statement. You list the statistic right after the variable name. To tell TABULATE that the statistic MEAN should be applied to the variable RENT, you use an asterisk to link the variable name to the statistic keyword. The asterisk is a TABULATE *operator*. Just as you use an asterisk as an operator when you want to multiply 2 by 3 (2*3), you use an asterisk when you want to apply a statistic to a variable.

```
PROC TABULATE DATA=TEMP;
  VAR RENT;
  TABLE RENT*MEAN;
RUN;
```

The output with the new statistic is shown in below. Note that the variable name at the top of the column heading has remained unchanged. However, the statistic name that is shown in the second line of the heading now says "MEAN." And the value shown in the table cell has changed from the sum to the mean.

```
„ƒƒƒƒƒƒƒƒƒƒƒƒ†
,     RENT     ,
‡ƒƒƒƒƒƒƒƒƒƒƒƒ‰
,     MEAN     ,
‡ƒƒƒƒƒƒƒƒƒƒƒƒ‰
,     1051.42,
ŠƒƒƒƒƒƒƒƒƒƒƒƒŒ
```

➢ ADDING ANOTHER STATISTIC

Each of the tables shown so far was useful, but the power of PROC TABULATE comes from being able to combine several statistics and/or several variables in the same table. TABULATE does this by letting you specify a series of "tables" within a single large table. We're going to add a "table" showing the number of observations to our table showing the mean rent.

The first part of our combined table is the code we used before to compute mean rent.

```
PROC TABULATE DATA=TEMP;
  VAR RENT;
  TABLE RENT*MEAN;
RUN;
```

Next, we can add similar code to our TABLE statement to get the number of observations. To add this statistic to the first table, all you do is combine the code for the mean ("RENT*MEAN") with the code you would used to get the number of observations ("RENT*N"). The code for the two "tables" is combined by using a space between the two statements. The space operator tells TABULATE that you want to add another column to your table.

```
PROC TABULATE DATA=TEMP;
  VAR RENT;
  TABLE RENT*N RENT*MEAN;
RUN;
```

The resulting table is shown below.

```
„ƒƒƒƒƒƒƒƒƒƒƒƒ…ƒƒƒƒƒƒƒƒƒƒƒƒ†
,     RENT     ,     RENT     ,
‡ƒƒƒƒƒƒƒƒƒƒƒƒˆƒƒƒƒƒƒƒƒƒƒƒƒ‰
,      N       ,     MEAN     ,
‡ƒƒƒƒƒƒƒƒƒƒƒƒˆƒƒƒƒƒƒƒƒƒƒƒƒ‰
,     128.00,     1051.42,
Šƒƒƒƒƒƒƒƒƒƒƒƒ‹ƒƒƒƒƒƒƒƒƒƒƒƒŒ
```

Note that the additional statistic is shown as an additional column in the table. When SAS is creating a one-dimensional table, additional variables, statistics, and categories are always added as new columns.

➢ USING PARENTHESES

While we're just building a simple table, other tables can get complex in a hurry. To keep your table code easy to read, it's helpful to simplify it as much as possible. One thing you can do is use parentheses to avoid repeating elements in row or column definitions.

For example, instead of defining the table statement like this:

```
TABLE RENT*N RENT*MEAN;
```

It can be defined like this:

```
TABLE RENT*(N MEAN);
```

The resulting table is shown below.

```
„ƒƒƒƒƒƒƒƒƒƒƒƒƒƒƒƒƒƒƒƒƒƒƒƒ†
,           RENT           ,
‡ƒƒƒƒƒƒƒƒƒƒƒƒ…ƒƒƒƒƒƒƒƒƒƒƒƒ‰
,     N     ,     MEAN     ,
‡ƒƒƒƒƒƒƒƒƒƒƒƒˆƒƒƒƒƒƒƒƒƒƒƒƒ‰
,     128.00,     1051.42,
Šƒƒƒƒƒƒƒƒƒƒƒƒ‹ƒƒƒƒƒƒƒƒƒƒƒƒŒ
```

Note that not only is the table definition easier to read, but the table headings have also been simplified. Now the "RENT" label is not repeated over each column, but rather is listed once as an overall heading. Using parentheses will simplify both your table definitions and your output.

➢ ADDING A CLASSIFICATION VARIABLE

After seeing the tables we've built so far in this chapter, you're probably asking yourself, "Why use PROC TABULATE? Everything I've seen so far could be done with a PROC MEANS."

One answer to this question is classification variables. By specifying a variable to categorize your data, you can produce a concise table that shows values for various subgroups in your data. For example, wouldn't it be more interesting to look at mean rent if it were broken down by city? [1]

To break down rent by city, we will use city as a classification variable. Just as we used a VAR statement to identify our *analysis* variable, we use a CLASS statement to identify a *classification* variable. By putting the variable CITY in a CLASS statement, we are telling TABULATE that the variable will be used to identify categories of the data.

The other thing we have to do to our code is tell TABULATE where to put the classification variable CITY in the table. We do this by again using the asterisk operator. By adding another asterisk to the end of the

[1] The sample data in this paper is from an informal survey of apartment rents in three cities: Portland, Oregon, where the author used to live; San Francisco, where the author currently lives; and Indianapolis, since that's where the paper will be presented.

TABLE statement, and following it with the variable name CITY, TABULATE knows that CITY will be used to categorize the mean values of RENT.

```
PROC TABULATE DATA=TEMP;
  CLASS CITY;
  VAR RENT;
  TABLE RENT*MEAN*CITY;
RUN;
```

The resulting table is shown below. Now the column headings have changed. The variable name RENT and the statistic name MEAN are still there, but under the statistic label there are now three columns. Each column is headed by the variable label "CITY" and the category name Portland," "San Francisco," and "Indianapolis." The values shown in the table cells now represent subgroup means.

➢ TWO-DIMENSIONAL TABLES

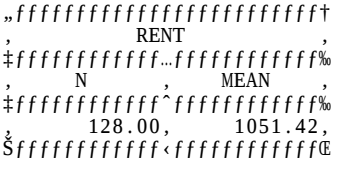
You probably noticed that our example table is not very elegant in appearance. That's because it only takes advantage of one dimension. It has multiple columns, but only one row. It is much more efficient to build tables that have multiple rows and multiple columns. You can fit more information on a page, and the table looks better, too.

The easiest way to build a two-dimensional table is to build it one dimension at a time. First we'll build the columns, and then we'll add the rows.

For this first table, we'll keep things simple. This is the table we built in a previous example. It has two columns: one showing the number of observations for RENT and another showing the mean of RENT.

```
PROC TABULATE DATA=TEMP;
  VAR RENT;
  TABLE RENT*(N MEAN);
RUN;
```

This table is shown below.



To turn this table into a two-dimensional table, we will add another variable to the TABLE statement. In this case, we want to add rows that show the N and MEAN of RENT for different sizes of apartments.

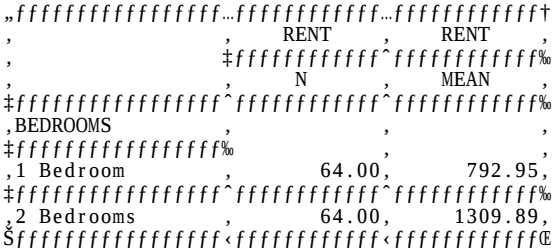
To add another dimension to the table, you use a comma as an operator. All you do is put a comma between the old part of the TABLE statement and the new part of the TABLE statement.

If a TABLE statement has no commas, then it is assumed that the variables and statistics are to be created as columns. If a TABLE statement has two parts, separated by a comma, then TABULATE builds a two-dimensional table using the first part of the TABLE statement as the rows and the second part of the TABLE statement as the columns.

So to get a table with rent as the columns and number of bedrooms as the rows, we just need to add a comma and the variable BEDROOMS. Since we want to add BEDROOMS as a row, we list it before the rest of the TABLE statement. If we wanted to add it as a column, we'd add it to the end of the TABLE statement.

```
PROC TABULATE DATA=TEMP;
  VAR RENT;
  CLASS BEDROOMS;
  TABLE BEDROOMS, RENT*N RENT*MEAN;
RUN;
```

This table is shown below.



By the way, there is a limit to which variables you can use in a two-dimensional table. You can't have a cross-tabulation of two analysis variables. A two-dimensional table must have at least one classification variable (i.e., you must have a CLASS statement). If you think about it, this makes sense. A table of mean rent by mean bedrooms would be meaningless, but a table of mean rent by categories of bedrooms makes perfect sense.

➢ ADDING CLASSIFICATION VARIABLES ON BOTH DIMENSIONS

The previous example showed how to reformat a table from one to two dimensions, but it did not show the true power of two-dimensional tables. With two dimensions, you can classify your statistics by two different variables at the same time.

To do this, you put one classification variable in the row dimension and one classification in the column dimension. The previous example had bedrooms displayed in rows, and rent as the column variable. In this new table, we will add city as an additional column variable. Instead

of just displaying the MEAN of the variable RENT for each number of bedrooms, we will display the statistic broken down city.

So in the following code, we leave BEDROOMS as the row variable, and we leave RENT in the column dimension. The only change is to add CITY to the column dimension using the asterisk operator. This tells TABULATE to break down each of the column elements into categories by city.

```
PROC TABULATE DATA=TEMP;
  VAR RENT;
  CLASS BEDROOMS CITY;
  TABLE BEDROOMS, RENT*CITY*MEAN;
RUN;
```

This table is shown below. Notice how the analysis variable RENT remains as the column heading, and MEAN remains as the statistic, but now there are additional column headings to show the three categories of CITY.

	RENT		
	CITY		
	Portland	San Francisco	Indianapolis
	MEAN	MEAN	MEAN
BEDROOMS			
1 Bedroom	620.63	1284.00	608.20
2 Bedrooms	1098.70	2099.59	923.75

➢ ADDING ANOTHER CLASSIFICATION VARIABLE

The previous example showed how to add a classification variable to both the rows and columns of a two-dimensional table. But you are not limited to just one classification per dimension. This next example will show how to display additional subgroups of the data.

In this case, we're going to add availability of off-street parking as an additional row classification. The variable is added to the CLASS statement and to the row dimension of the TABLE statement. It is added using a space as the operator, so we will get rows for number of bedrooms followed by rows for parking availability.

```
PROC TABULATE DATA=TEMP;
  VAR RENT;
  CLASS BEDROOMS CITY PARKING;
  TABLE BEDROOMS PARKING,
    RENT*CITY*MEAN;
RUN;
```

In the results shown below, you can see that we now have two two-dimensional "mini-tables" within a single table. First we have a table of rent by number of bedrooms and city, and then we have a table showing rent by parking availability and city.

	RENT		
	CITY		
	Portland	San Francisco	Indianapolis
	MEAN	MEAN	MEAN
BEDROOMS			
1 Bedroom	620.63	1284.00	608.20
2 Bedrooms	1098.70	2099.59	923.75
PARKING			
No	880.09	1479.00	636.50
Yes	845.63	1780.46	895.45

This ability to stack multiple mini-tables within a single table can be a powerful tool for delivering large quantities of information in a user-friendly format.

➢ NESTING THE CLASSIFICATION VARIABLES

So far all we have done is added additional "tables" to the bottom of our first table. We used the space operator between each of the row variables to produce stacked tables.

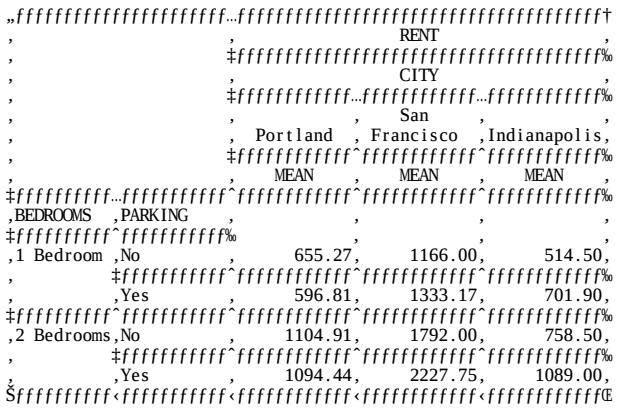
By using a series of row variables, we can explore a variety of relationships between the variables. In previous table, we could see how rent varies by number of bedrooms and city, and we could see how rent varies by parking availability and city, but we could not see how number of bedrooms and parking availability interacted to affect rent for each city.

But we can learn even more from our data. The power of TABULATE comes from being able to look at combinations of categories within a single table. In the following example, we will build a table to look at rent by city for combinations of number of bedrooms and parking availability.

This code is the same as we used for the last example. The only change is that in the row definition, the asterisk operator is used to show that we want to nest the two row variables. In other words, we want to see the breakdown of rents by parking availability within each category of number of bedrooms.

```
PROC TABULATE DATA=TEMP;
  VAR RENT;
  CLASS BEDROOMS CITY PARKING;
  TABLE BEDROOMS*PARKING,
    RENT*CITY*MEAN;
RUN;
```

As you can see in the table below, this code produces nested categories within the row headings. The row headings are now split into two columns. The first column shows number of bedrooms and the second shows parking availability. It is now easier to interpret the interaction of the two variables.



You can also reverse the order of the row variables to look at number of bedrooms by parking availability, instead of parking availability by number of bedrooms. All you do is move PARKING so that it comes before BEDROOMS. TABULATE always produces the nested rows in the order the variables are listed on the TABLE statement.

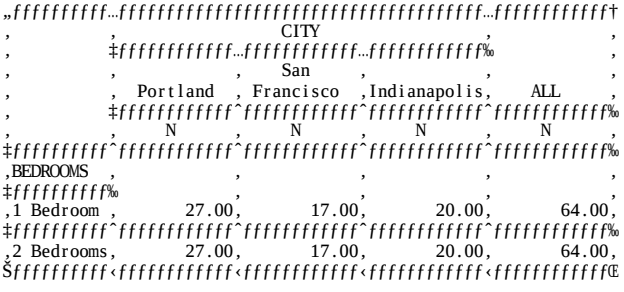
➢ ADDING TOTALS TO THE ROWS AND COLUMNS

As your tables get more complex, you can help make your tables more readable by adding row and column totals. Totals are quite easy to generate in TABULATE because you can use the ALL variable. This is a built in classification variable supplied by TABULATE that stands for "all observations." You do not have to list it in the CLASS statement because it is a classification variable by definition.

The following code produces a table similar to the previous example, but with the addition of row totals. The statistic is changed to N so that you can see how the totals work.

```
PROC TABULATE DATA=TEMP;
  CLASS BEDROOMS CITY;
  TABLE BEDROOMS, (CITY ALL)*N;
RUN;
```

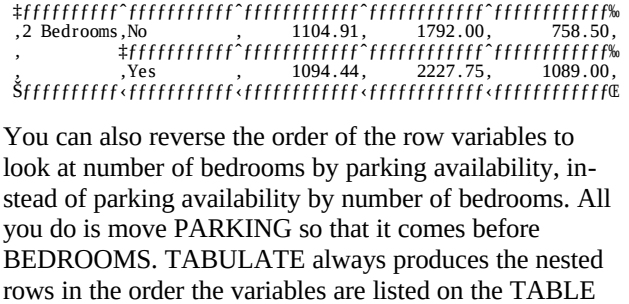
As you can see from the table below, the table now has row totals.



Not only can you use ALL to add row totals, but you can also use ALL to produce column totals. What you do is list ALL as an additional variable in the row definition of the TABLE statement. No asterisk is needed because we just want to add a total at the bottom of the table.

```
PROC TABULATE DATA=TEMP;
  CLASS BEDROOMS CITY;
  TABLE BEDROOMS ALL, CITY*N;
RUN;
```

The resulting table is shown below. Now there are overall totals for each city.



➢ THREE-DIMENSIONAL TABLES

Now that you have mastered two-dimensional tables, let's add a third dimension. You may be asking yourself: Three dimensions? How do you print a table shaped like a cube?

Actually, a three-dimensional table is not shaped like a cube. It looks like a two-dimensional table, except that it spans multiple pages. A one-dimensional table just has columns. A two-dimensional table has both columns and rows. A three-dimensional table is just a two-dimensional table that is repeated across multiple pages. Basically, you print a new page for each value of the page variable.

The hardest part about three-dimensional tables is making sense of the TABLE statement. So the best way to start is with the first two dimensions: the rows and columns. Once you've got that set up correctly, it's relatively easy to add the page variable to expand the table to multiple pages.

For our example, we're going to build another table of rent by city and number of bedrooms, and then we're going to add parking availability as the page variable. We'll end up with two pages of tables, the first page will be for apartments without off-street parking, and the second page will be for apartments with off-street parking.

Ignoring the third dimension for now, let's build the basic table. This table has rows showing the number of bedrooms, and columns showing rent by city. The code is as follows:

```
PROC TABULATE DATA=TEMP;
  CLASS BEDROOMS CITY;
  VAR RENT;
  TABLE BEDROOMS,
      (CITY ALL)*RENT*MEAN;
RUN;
```

At this point you should run the code and look the table over carefully to be sure you've got exactly what you want to see in your final table.

	CITY			
	Portland	San Francisco	Indianapolis	ALL
	RENT	RENT	RENT	RENT
	MEAN	MEAN	MEAN	MEAN
BEDROOMS				
1 Bedroom	620.63	1284.00	608.20	792.95
2 Bedrooms	1098.70	2099.59	923.75	1309.89

The only difference between this table and our final three-dimensional table is that right now, the table is showing results for both categories of parking availability combined. In the final tables, each page will have the results for just one of the two categories.

Assuming the two-dimensional table looks correct, we'll go on to adding the third dimension. Just as when we converted a one-dimensional table to a two-dimensional table, we add a new dimension with a comma operator in the TABLE statement. We just add the new variable PARKING to the existing TABLE statement with a comma to separate it from the row and column definitions.

You might think that the following code is the correct way to add the third dimension to the table statement:

```
TABLE BEDROOMS,
  (CITY ALL)*RENT*MEAN, PARKING;
```

However, what this would generate is a table with CITY as the rows, PARKING as the columns, and BEDROOMS as the pages! In order to add the third dimension to a table, you add it at the beginning of the table statements. Remember that this was also true when we added the second dimension to the table. We added rows to the columns by adding a row definition before the column definition.

The correct code for our table is:

```
PROC TABULATE DATA=TEMP;
  CLASS BEDROOMS CITY PARKING;
  VAR RENT;
  TABLE PARKING, BEDROOMS,
      (CITY ALL)*RENT*MEAN;
RUN;
```

The resulting tables are shown below. To save space, both tables are shown on a single page. In reality, TABULATE puts a page break between each of the tables, and the table for apartments without off-street parking would appear on a second page. Notice that each table was automatically given a title that defines the category it represents.

PARKING No

	CITY			
	Portland	San Francisco	Indianapolis	ALL
	RENT	RENT	RENT	RENT
	MEAN	MEAN	MEAN	MEAN
BEDROOMS				
1 Bedroom	655.27	1166.00	514.50	699.35
2 Bedrooms	1104.91	1792.00	758.50	1103.81

PARKING Yes

	CITY			
	Portland	San Francisco	Indianapolis	ALL
	RENT	RENT	RENT	RENT
	MEAN	MEAN	MEAN	MEAN
BEDROOMS				
1 Bedroom	596.81	1333.17	701.90	857.00
2 Bedrooms	1094.44	2227.75	1089.00	1450.89

➢ MAKING THE TABLE PRETTY

The preceding examples have shown how to create basic tables. They contained all of the needed information, but they were pretty ugly. The next thing you need to learn is a few tricks to clean up your tables.

For example, look at the following code and table:

```
PROC TABULATE DATA=TEMP;
  CLASS BEDROOMS CITY;
  VAR RENT;
  TABLE BEDROOMS,
      (CITY ALL)*RENT*MEAN;
RUN;
```

	CITY			
	Portland	San Francisco	Indianapolis	ALL
	RENT	RENT	RENT	RENT
	MEAN	MEAN	MEAN	MEAN
BEDROOMS				
1 Bedroom	620.63	1284.00	608.20	792.95
2 Bedrooms	1098.70	2099.59	923.75	1309.89

Notice how the totals column is titled "All." We can make this table more readable by changing that title to "Overall."

To do this, we just attach the label to the keyword in the TABLE statement with an equal sign. The new code reads as follows:

```
PROC TABULATE DATA=TEMP;
  CLASS BEDROOMS CITY;
  VAR RENT;
  TABLE BEDROOMS,
      (CITY ALL='Overall')*RENT*MEAN;
RUN;
```

This code produces the following output:

```
„ƒƒƒƒƒƒƒƒƒ…ƒƒƒƒƒƒƒƒƒƒƒƒƒƒƒƒƒƒƒƒƒƒƒƒƒƒƒƒƒƒƒƒƒƒƒƒƒƒ…ƒƒƒƒƒƒƒƒƒƒƒ†
,         ,                 CITY                 ,           ,
,         ‡ƒƒƒƒƒƒƒƒƒƒƒ…ƒƒƒƒƒƒƒƒƒƒƒ…ƒƒƒƒƒƒƒƒƒƒƒ‰           ,
,         ,           ,    San    ,           ,           ,
,         , Portland  , Francisco ,Indianapolis,  Overall  ,
,         ‡ƒƒƒƒƒƒƒƒƒƒƒˆƒƒƒƒƒƒƒƒƒƒƒˆƒƒƒƒƒƒƒƒƒƒƒˆƒƒƒƒƒƒƒƒƒƒƒ‰
,         ,   RENT    ,   RENT    ,   RENT    ,   RENT    ,
,         ‡ƒƒƒƒƒƒƒƒƒƒƒˆƒƒƒƒƒƒƒƒƒƒƒˆƒƒƒƒƒƒƒƒƒƒƒˆƒƒƒƒƒƒƒƒƒƒƒ‰
,         ,   MEAN    ,   MEAN    ,   MEAN    ,   MEAN    ,
‡ƒƒƒƒƒƒƒƒƒˆƒƒƒƒƒƒƒƒƒƒƒˆƒƒƒƒƒƒƒƒƒƒƒˆƒƒƒƒƒƒƒƒƒƒƒˆƒƒƒƒƒƒƒƒƒƒƒ‰
,BEDROOMS ,           ,           ,           ,           ,
‡ƒƒƒƒƒƒƒƒƒ‰           ,           ,           ,           ,
,1 Bedroom ,    620.63,    1284.00,     608.20,     792.95,
‡ƒƒƒƒƒƒƒƒƒˆƒƒƒƒƒƒƒƒƒƒƒˆƒƒƒƒƒƒƒƒƒƒƒˆƒƒƒƒƒƒƒƒƒƒƒˆƒƒƒƒƒƒƒƒƒƒƒ‰
,2 Bedrooms,   1098.70,    2099.59,     923.75,    1309.89,
Šƒƒƒƒƒƒƒƒƒ‹ƒƒƒƒƒƒƒƒƒƒƒ‹ƒƒƒƒƒƒƒƒƒƒƒ‹ƒƒƒƒƒƒƒƒƒƒƒ‹ƒƒƒƒƒƒƒƒƒƒƒŒ
```

Another thing that could be improved about this table is getting rid of the excessive column headings. This table is four lines deep in headings. For starters, we can get rid of the label “City.” With values like “San Francisco,” “Portland,” and “Indianapolis,” it’s obvious that this table is referring to cities. We don’t need the extra label.

To get rid of it, we attach a blank label. A blank label is two quotes with a single blank space in between. The blank label is added the same way we added the “Total” label in the last example. The blank label is attached to the variable using an equal sign.

```
PROC TABULATE DATA=TEMP;
  CLASS BEDROOMS CITY;
  VAR RENT;
  TABLE BEDROOMS,
      (CITY=' ' ALL='Overall')*
      RENT*MEAN;
RUN;
```

The revised output is shown below.

```
„ƒƒƒƒƒƒƒƒƒ…ƒƒƒƒƒƒƒƒƒƒƒ…ƒƒƒƒƒƒƒƒƒƒƒ…ƒƒƒƒƒƒƒƒƒƒƒ…ƒƒƒƒƒƒƒƒƒƒƒ†
,         ,           ,    San    ,           ,           ,
,         , Portland  , Francisco ,Indianapolis,  Overall  ,
,         ‡ƒƒƒƒƒƒƒƒƒƒƒˆƒƒƒƒƒƒƒƒƒƒƒˆƒƒƒƒƒƒƒƒƒƒƒˆƒƒƒƒƒƒƒƒƒƒƒ‰
,         ,   RENT    ,   RENT    ,   RENT    ,   RENT    ,
,         ‡ƒƒƒƒƒƒƒƒƒƒƒˆƒƒƒƒƒƒƒƒƒƒƒˆƒƒƒƒƒƒƒƒƒƒƒˆƒƒƒƒƒƒƒƒƒƒƒ‰
,         ,   MEAN    ,   MEAN    ,   MEAN    ,   MEAN    ,
‡ƒƒƒƒƒƒƒƒƒˆƒƒƒƒƒƒƒƒƒƒƒˆƒƒƒƒƒƒƒƒƒƒƒˆƒƒƒƒƒƒƒƒƒƒƒˆƒƒƒƒƒƒƒƒƒƒƒ‰
,BEDROOMS ,           ,           ,           ,           ,
‡ƒƒƒƒƒƒƒƒƒ‰           ,           ,           ,           ,
,1 Bedroom ,    620.63,    1284.00,     608.20,     792.95,
‡ƒƒƒƒƒƒƒƒƒˆƒƒƒƒƒƒƒƒƒƒƒˆƒƒƒƒƒƒƒƒƒƒƒˆƒƒƒƒƒƒƒƒƒƒƒˆƒƒƒƒƒƒƒƒƒƒƒ‰
,2 Bedrooms,   1098.70,    2099.59,     923.75,    1309.89,
Šƒƒƒƒƒƒƒƒƒ‹ƒƒƒƒƒƒƒƒƒƒƒ‹ƒƒƒƒƒƒƒƒƒƒƒ‹ƒƒƒƒƒƒƒƒƒƒƒ‹ƒƒƒƒƒƒƒƒƒƒƒŒ
```

This looks much better, but there are still too many column headings. We can get rid of two more. Notice how each column is headed by “RENT” and “MEAN.”

We can make these labels go away by setting them to a blank label as in the previous example. Also, the row heading “Number of Bedrooms” is not needed, since the categories of “1 Bedroom” and “2 Bedrooms” are self-explanatory. So the variable BEDROOMS is also assigned a blank label. The revised code is shown below.

```
PROC TABULATE DATA=TEMP;
  CLASS BEDROOMS CITY;
  VAR RENT;
  TABLE BEDROOMS=' ',
      (CITY=' ' ALL='Overall')*
      RENT=' '*MEAN=' ';
RUN;
```

However, you can see in the following output that we have a problem:

```
„ƒƒƒƒƒƒƒƒƒ…ƒƒƒƒƒƒƒƒƒƒƒ…ƒƒƒƒƒƒƒƒƒƒƒ…ƒƒƒƒƒƒƒƒƒƒƒ…ƒƒƒƒƒƒƒƒƒƒƒ†
,         ,           ,    San    ,           ,           ,
,         , Portland  , Francisco ,Indianapolis,  Overall  ,
‡ƒƒƒƒƒƒƒƒƒˆƒƒƒƒƒƒƒƒƒƒƒˆƒƒƒƒƒƒƒƒƒƒƒˆƒƒƒƒƒƒƒƒƒƒƒˆƒƒƒƒƒƒƒƒƒƒƒ‰
,1 Bedroom ,    620.63,    1284.00,     608.20,     792.95,
‡ƒƒƒƒƒƒƒƒƒˆƒƒƒƒƒƒƒƒƒƒƒˆƒƒƒƒƒƒƒƒƒƒƒˆƒƒƒƒƒƒƒƒƒƒƒˆƒƒƒƒƒƒƒƒƒƒƒ‰
,2 Bedrooms,   1098.70,    2099.59,     923.75,    1309.89,
Šƒƒƒƒƒƒƒƒƒ‹ƒƒƒƒƒƒƒƒƒƒƒ‹ƒƒƒƒƒƒƒƒƒƒƒ‹ƒƒƒƒƒƒƒƒƒƒƒ‹ƒƒƒƒƒƒƒƒƒƒƒŒ
```

If we take the RENT and MEAN labels away, then there is no label in the table to describe the analysis variable or statistic.

We could add a title above the table to hold this information, but there’s a better way. Notice how in all of our tables there’s a big empty box above the rows and to the left of the column headings. This space is available to us. The following code uses the BOX= option to hold a label describing our table.

```
PROC TABULATE DATA=TEMP;
  CLASS BEDROOMS CITY;
  VAR RENT;
  TABLE BEDROOMS,
      (CITY=' ' ALL='Overall')*
      RENT=' '*MEAN=' '
      / BOX='Average Rent';
RUN;
```

The output is shown below:

```
„ƒƒƒƒƒƒƒƒƒ…ƒƒƒƒƒƒƒƒƒƒƒ…ƒƒƒƒƒƒƒƒƒƒƒ…ƒƒƒƒƒƒƒƒƒƒƒ…ƒƒƒƒƒƒƒƒƒƒƒ†
,Average  ,           ,    San    ,           ,           ,
,Rent     , Portland  , Francisco ,Indianapolis,  Overall  ,
‡ƒƒƒƒƒƒƒƒƒˆƒƒƒƒƒƒƒƒƒƒƒˆƒƒƒƒƒƒƒƒƒƒƒˆƒƒƒƒƒƒƒƒƒƒƒˆƒƒƒƒƒƒƒƒƒƒƒ‰
,BEDROOMS ,           ,           ,           ,           ,
‡ƒƒƒƒƒƒƒƒƒ‰           ,           ,           ,           ,
,1 Bedroom ,    620.63,    1284.00,     608.20,     792.95,
‡ƒƒƒƒƒƒƒƒƒˆƒƒƒƒƒƒƒƒƒƒƒˆƒƒƒƒƒƒƒƒƒƒƒˆƒƒƒƒƒƒƒƒƒƒƒˆƒƒƒƒƒƒƒƒƒƒƒ‰
,2 Bedrooms,   1098.70,    2099.59,     923.75,    1309.89,
Šƒƒƒƒƒƒƒƒƒ‹ƒƒƒƒƒƒƒƒƒƒƒ‹ƒƒƒƒƒƒƒƒƒƒƒ‹ƒƒƒƒƒƒƒƒƒƒƒ‹ƒƒƒƒƒƒƒƒƒƒƒŒ
```

At this point the table is looking pretty good. There’s just one more thing we should do to the table. Since the numbers being reported in the table are rents, it would make the table easier to read if they were formatted with dollar signs. Also, we can round these off to even dollars, that’s enough precision for this table.

To change the format of the table cells, we use the FORMAT= option on the TABLE statement. You can use this to apply any valid SAS format. The revised code calls for values to be formatted with dollar signs and commas, and eliminates the display of decimal spaces. The width of 12 was chosen because the widest column label ("Indianapolis") is 12 characters wide. When you specify a format for the table values, you are also specifying the width of the column that holds that value.

```
PROC TABULATE DATA=TEMP
     FORMAT=DOLLAR12.;
  CLASS BEDROOMS CITY;
  VAR RENT;
  TABLE BEDROOMS=' ',
     (CITY=' ' ALL='Overall')*
     RENT=' '*MEAN=' '
     / BOX='Average Rent';
RUN;
```

The revised output is shown below:

```
„ƒƒƒƒƒƒƒƒƒƒ…ƒƒƒƒƒƒƒƒƒƒƒƒ…ƒƒƒƒƒƒƒƒƒƒƒƒ…ƒƒƒƒƒƒƒƒƒƒƒƒ…ƒƒƒƒƒƒƒƒƒƒƒƒ†
,Average   ,            ,    San     ,            ,            ,
,Rent      , Portland   , Francisco  ,Indianapolis, Overall    ,
‡ƒƒƒƒƒƒƒƒƒƒˆƒƒƒƒƒƒƒƒƒƒƒƒˆƒƒƒƒƒƒƒƒƒƒƒƒˆƒƒƒƒƒƒƒƒƒƒƒƒˆƒƒƒƒƒƒƒƒƒƒƒƒ‰
,1 Bedroom ,       $621,      $1,284,        $608,        $793,
‡ƒƒƒƒƒƒƒƒƒƒˆƒƒƒƒƒƒƒƒƒƒƒƒˆƒƒƒƒƒƒƒƒƒƒƒƒˆƒƒƒƒƒƒƒƒƒƒƒƒˆƒƒƒƒƒƒƒƒƒƒƒƒ‰
,2 Bedrooms,     $1,099,      $2,100,        $924,      $1,310,
Šƒƒƒƒƒƒƒƒƒƒ‹ƒƒƒƒƒƒƒƒƒƒƒƒ‹ƒƒƒƒƒƒƒƒƒƒƒƒ‹ƒƒƒƒƒƒƒƒƒƒƒƒ‹ƒƒƒƒƒƒƒƒƒƒƒƒŒ
```

➢ CREATING HTML OUTPUT

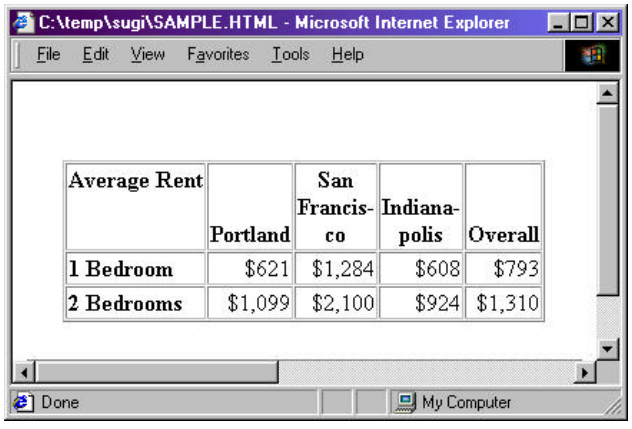
Once you know how to create TABULATE tables, it is a simple matter to turn them into HTML tables that can be posted on your web site.

Using Version 6, all you have to do is download and install the HTML conversion macros that are available on the SAS web site (www.sas.com). Then, you just add a macro call before and after your TABULATE code and SAS will generate the HTML output for you.

The Version 6 code is as follows:

```
%TAB2HTM (CAPTURE=ON, RUNMODE=B);
OPTIONS FORMCHAR='8283848586878889 8A8B8C'X;
PROC TABULATE DATA=TEMP
     FORMAT=DOLLAR12.;
  CLASS BEDROOMS CITY;
  VAR RENT;
  TABLE BEDROOMS=' ',
     (CITY=' ' ALL='Overall')*
     RENT=' '*MEAN=' '
     / BOX='Average Rent';
RUN;
%TAB2HTM(CAPTURE=OFF, RUNMODE=B,
  OPENMODE=REPLACE, HTMLFILE=SAMPLE.HTML);
```

The output is shown below:

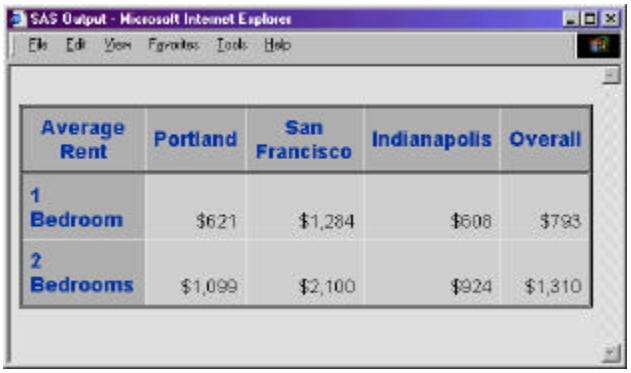


Average Rent	Portland	San Francis-co	Indiana-polis	Overall
1 Bedroom	$621	$1,284	$608	$793
2 Bedrooms	$1,099	$2,100	$924	$1,310

Under Versions 7-8, the process of outputting a TABULATE table to HTML is nearly identical. Instead of calling the macro before and after your code, you call on the Output Delivery System (ODS).

```
ODS HTML BODY='SAMPLE.HTML';
PROC TABULATE DATA=TEMP
     FORMAT=DOLLAR12.;
  CLASS BEDROOMS CITY;
  VAR RENT;
  TABLE BEDROOMS=' ',
     (CITY=' ' ALL='Overall')*
     RENT=' '*MEAN=' '
     / BOX='Average Rent';
RUN;
ODS HTML CLOSE;
```

The output is shown below:



Average Rent	Portland	San Francisco	Indianapolis	Overall
1 Bedroom	$621	$1,284	$608	$793
2 Bedrooms	$1,099	$2,100	$924	$1,310

You can see that the result is nearly identical to the Version 6 table. The only difference is that the Version 7 table uses more colors, and is more stylish.

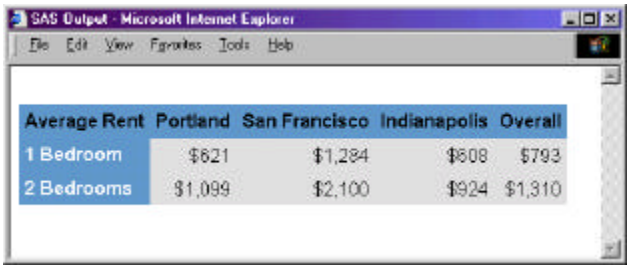
➢ CHANGING THE STYLE

If you're using Version 7 or 8, you also have the option of changing the style. By adding a STYLE= option to your ODS statement, you can change from the default style to one of the following styles. The resulting output is shown below the code for each style

```
ODS HTML BODY='SAMPLE.HTML'
STYLE=XXX;
```

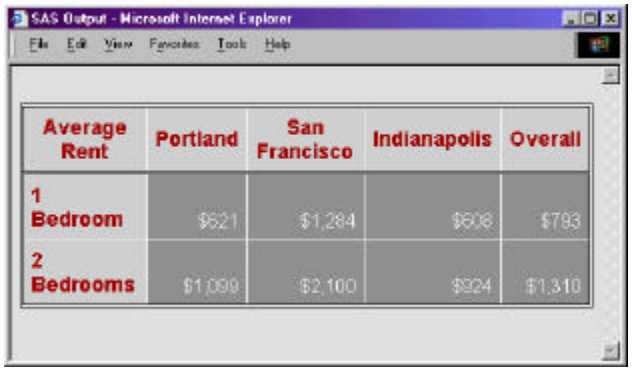
The following examples show a few of the styles that ship with Versions 7-8.

STYLE=BARRETTSBLUE

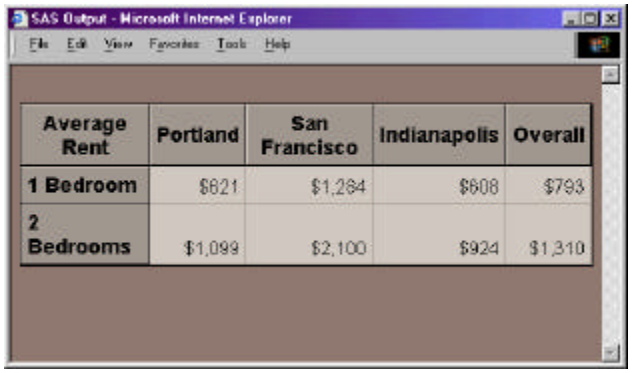


Average Rent	Portland	San Francisco	Indianapolis	Overall
1 Bedroom	$621	$1,284	$608	$793
2 Bedrooms	$1,099	$2,100	$924	$1,310

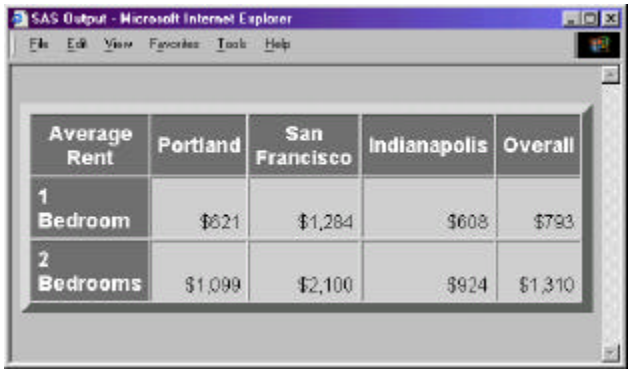
STYLE=BRICK



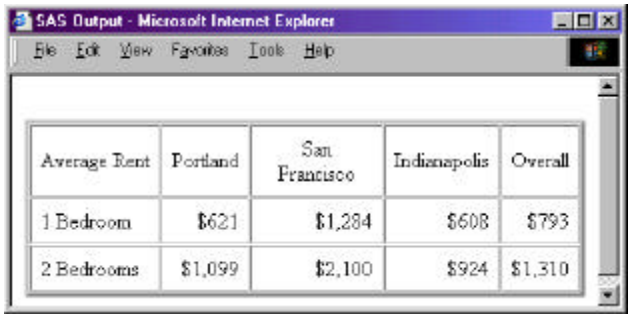
STYLE=BROWN



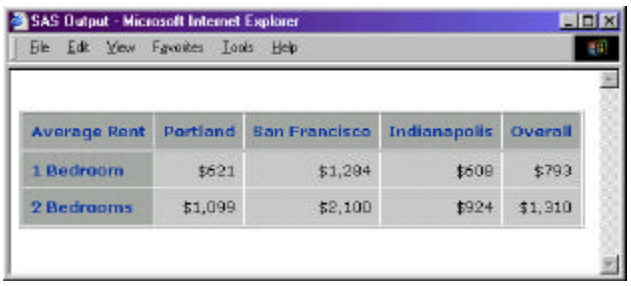
STYLE=D3D



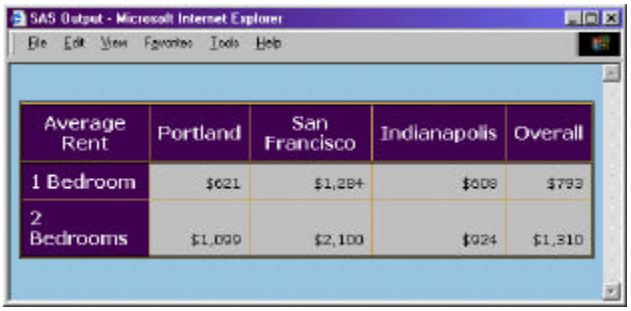
STYLE=MINIMAL



STYLE=STATDOC



STYLE=THEME



➢ CONCLUSIONS

At this point, you should be comfortable with the basics of producing a table using PROC TABULATE. You should be able to produce a simple table with totals, be able to clean it up a bit, and be able to create HTML output.

This should be enough to get you going producing tables with your own data. And now that you're more comfortable with the procedure, you should be able to use the TABULATE manual and other books and papers to learn more advanced techniques.

➢ ACKNOWLEDGEMENTS

SAS is a registered trademark or trademark of SAS Institute Inc. in the USA and other countries. ® indicates USA registration.

➢ CONTACTING THE AUTHOR

Please direct any questions or feedback to the author at:

Ischemia Research & Education Foundation
250 Executive Park Blvd, Suite 3400
San Francisco, CA 94134

E-mail: leh@iref.org