DM06-2012

SAS® Data Management Techniques:

Cleaning and transforming data for delivery of analytic datasets

Christopher W. Schacherer, Clinical Data Management Systems, LLC

ABSTRACT

The analytic capabilities of SAS® software are unparalleled. Similarly, the ability of the Output Delivery System® to produce an endless array of creative, high-quality reporting solutions is the envy of many analysts using competing tools. But beneath all of the glitz and glitter is the dirty work of cleaning, managing, and transforming raw source data and reliably delivering analytic datasets that accurately reflect the processes being analyzed. Although a basic understanding of DATA step processing and PROC SQL is assumed, the present work provides examples of both basic data management techniques for profiling data as well as transformation techniques that are commonly utilized in the creation of analytic data products. Finally, examples of techniques for automating the generation and delivery of production-quality, enterprise-level datasets are provided.

INTRODUCTION

To a greater extent than ever before, organizations of all types consider their operational data to be one of their greatest assets. By understanding their operations and benchmarking performance against their competitors, organizations are able to capitalize on their strengths and correct their short-comings to remain competitive in their respective market. The ability to make good decisions in a timely manner is predicated on the assumption that the analytic datasets used for these analyses are both reliable and valid. Assuring that this assumption is correct is the responsibility of everyone who touches the data, but usually falls most heavily on a relatively small group of data managers tasked with making sure analytic datasets are always ready on time (day-to-day, month-to-month, quarter-to-quarter), are always easy to understand and use, and always accurately reflect your organization's operations. Especially for these hardy few, but also for everyone in the data value-chain, the current work describes several techniques commonly used to integrate new analytic datasets into your production environment. Regardless of whether you are working in a one-person data management shop or a large, sophisticated business intelligence unit, the approaches outlined below provide practical tips for commonly-encountered challenges in the creation of a "data product".

DATA QUALITY

Arguably, the most widely debated aspect of data management is the concept of data quality. Even before you can effectively argue about which metrics of data quality have the greatest impact on analyses (e.g., missing/unclassified data-points, orphan records, etc.), you need to have a higher-level understanding of what you mean by "data quality". Some people would argue that a dataset is of high quality if the data accurately reflect the data points as they exist in the source system from which the data were extracted. Others argue that definitions of quality are more elusive—having to do with the wants and needs of customers or the goodness of fit between the data and its desired use. Clearly there is merit to both arguments; you certainly want to be able to trace your data back through the various systems and processes that gave rise to it, but at the end of the day, you want some measure of assurance that decisions made on the basis of these data are grounded in an accurate depiction of the processes that you are studying and are not littered with artifacts of the systems used to store and manage the data. Short of studying every business process in your organization and mapping it, yourself, to each data point (though, someone in your organization should have done this at some point and documented it) the best you can do is to interrogate the data in an attempt to discover any anomalies that might threaten their validity. The usual starting point for this discovery process is the data dictionary.

DATA DICTIONARIES

If you are fortunate, you work in an organization with an effective, robust data governance system that has tackled the issue of data quality and its important component, the metadata repository. If not, at the very least when you begin discussing the incorporation of new source data into your production analytic environment one of the first points of discussion about the source data should be the need for a complete and accurate data dictionary. Whether it is a wiki, static HTML page, or PDF, this document should contain at a minimum (in addition to the name and contact information of the party responsible for its content) the following information for each variable in the source dataset(s):

<table>
<tr><th>Component</th><th>Description</th></tr>
<tr><td>Variable Name</td><td>The name of each variable in the dataset. If the dataset is delivered without a record/data line indicating the name of each variable, you at least need an entry identifying the variable's position within each data record.</td></tr>
<tr><td>Variable Description</td><td>Each variable should be described in terms of how it characterizes an individual record in the dataset (e.g., "weight" might be described as "The patient's weight in kilograms".). Of course, the more rich these descriptions, the better. For example, "the patient's weight at the beginning of the current course of therapy" may mean something very different than "the patient's weight at diagnosis".</td></tr>
<tr><td>Data Type/Length/Format</td><td>An extremely important characteristic of each variable in your dataset is its data type—whether the variable is "numeric", a "character string", "date", etc. As discussed later, these descriptions may or may not reflect the actual data received and such discrepancies are one of the most common challenges data managers face. In addition to the "type" of data, the length and format (or, precision, for numeric data) can have important implications for delivering a high-quality data product.</td></tr>
<tr><td>Nulls Allowed?</td><td>It is important to understand (from the provider of the data) whether null values are allowed by the source system. If null (or missing) values are declared to not be allowed and you find them in your data, this is an important event to communicate to the data provider. Conversely, if you know that null values are allowed, you can keep your SAS log clean by programming data transformations in a manner that keeps your program from generating repetitive notes/warnings related to missing values.</td></tr>
<tr><td>Acceptable Values & Code/Label Pairs</td><td>For numeric values that provide measurements (e.g., $12.51 spent per transaction, 125 kilograms, 22 respirations/minute) it is important to know the acceptable range of values for those measurements. For example, a "patient weight" variable might have the acceptable value range "0.10 – 999.99, 9999.00 = Missing"—indicating that values between 1000.00 and 9998.99 are not valid and that for the purposes of calculating mean weight (for example), occurrences of weight with the value "9999.00" should be converted to missing (.) before computing the statistic.

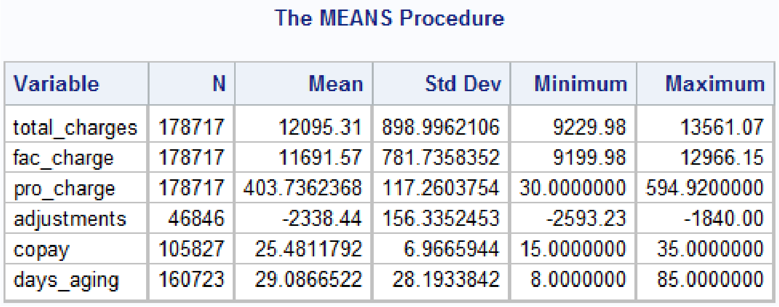
For numeric variables associated with a coded value (e.g., 0 = 'No', 1 = 'Yes') and character variables that indicate categories (e.g., 'M'=Male, 'F'=Female) it is critical to have a list of the code/label pairs.</td></tr>
<tr><td>Start Position (for fixed-width files)</td><td>Though less frequently encountered now than in the past, many of us still deal frequently with files in fixed-width format. Data dictionaries for these files should include at least the starting position and the length of the variable. Frequently, they also contain the ending position as well. For example, in describing patient height and weight, you might find that the data dictionary contains the following entries which will be vital to writing the INPUT statement necessary for reading the data into SAS:
<table>
<tr><th>Variable</th><th>Length</th><th>Start</th><th>End</th></tr>
<tr><td>Weight</td><td>7</td><td>17</td><td>23</td></tr>
<tr><td>Height</td><td>2</td><td>25</td><td>26</td></tr>
</table>
</td></tr>
</table>

DATA PROFILING

Having been provided an adequate data dictionary, you are ready to start profiling the new source data. Data profiling is the analysis of the data for the purpose of (a) confirming that the data conform to the constraints described in the data dictionary and (b) discovering the ways in which the data—though not violating the constraints in the data dictionary—might provide an erroneous or misleading view of the process that you are studying. The good news, according to DeMets (1997), is that truly random errors rarely have an impact on the outcome of our analyses; the problem is that we do not know which of the errors we are looking at are due to chance and which are the result of some systematic error (or fraud) in the upstream data process. The latter types of errors definitely need to be identified and corrected if your analyses and reports are to be valid.

Numeric Variables - Range Checks. – The most obvious check to perform on numeric measurement variables (i.e., those numeric variables that are intended to act as ordinal, interval, or ratio measures of something—dollars spent or billed, distances traveled, heartbeats per minute, etc.) is to confirm that their minimum and maximum values are within the range specified by your data dictionary. In the following example, the variables total_charges, pro_charge, fac_charge, copay, adjustments, and days_aging are all numeric variables that provide a measurement of something in a healthcare claims dataset. According to the data dictionary you received from your system administrator, all six variables should have values between 0 and 99999999. One quick way to confirm that this is the case is to run the MEANS PROCEDURE against these variables to confirm this assertion.

```
PROC MEANS DATA=hca.claims;
 VAR total_charges pro_charge fac_charge copay adjustments days_aging;
RUN;
```

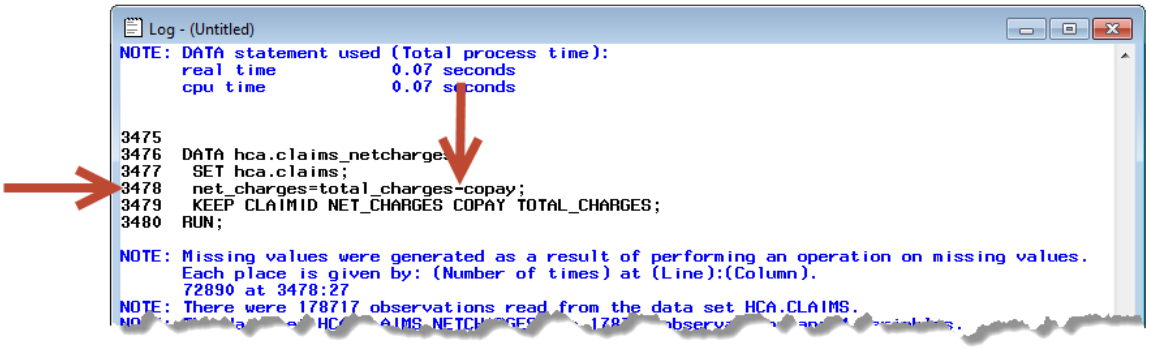


The MEANS Procedure

Variable	N	Mean	Std Dev	Minimum	Maximum
total_charges	178717	12095.31	898.9962106	9229.98	13561.07
fac_charge	178717	11691.57	781.7358352	9199.98	12966.15
pro_charge	178717	403.7362368	117.2603754	30.0000000	594.9200000
adjustments	46846	-2338.44	156.3352453	-2593.23	-1840.00
copay	105827	25.4811792	6.9665944	15.0000000	35.0000000
days_aging	160723	29.0866522	28.1933842	8.0000000	85.0000000

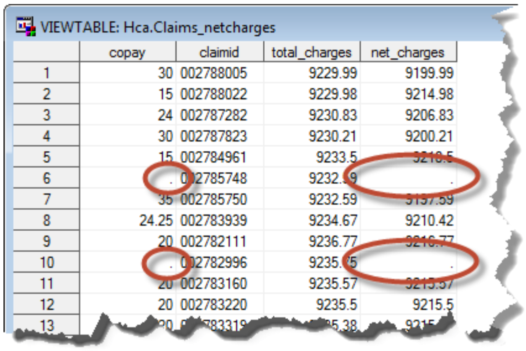
```
NOTE: There were 178717 observations read from the data set HCA.CLAIMS.
```

This simple check tells you a lot about the "claims" dataset. First, from the log file note you see that there were 178,717 records in the dataset. Immediately you are able to confirm that there are no missing values in the first three variables (total_charges, fac_charge, and pro_charge). Further, because the minimum values for these first three variables are all greater than zero, the assertion that all values are between 0 and 99999999 holds true. However, in looking at the results for "adjustments", you see that the maximum value is a negative number. Clearly, the minimum value is not zero. Furthermore, the "N" of adjustments, copay, and days_aging reveal that there are missing values for these variables. This is important information to have before beginning transformations of the data that involve calculations involving these variables. What would happen, for example if you calculated the net charges for the claims in this dataset by simply subtracting the copay amount from the total_charges?



```
NOTE: DATA statement used (Total process time):
      real time           0.07 seconds
      cpu time            0.07 seconds


3475
3476  DATA hca.claims_netcharges;
3477   SET hca.claims;
3478   net_charges=total_charges-copay;
3479   KEEP CLAIMID NET_CHARGES COPAY TOTAL_CHARGES;
3480  RUN;

NOTE: Missing values were generated as a result of performing an operation on missing values.
      Each place is given by: (Number of times) at (Line):(Column).
      72890 at 3478:27
NOTE: There were 178717 observations read from the data set HCA.CLAIMS.
```

As seen in the log (above) and in the dataset (below), the result of performing a calculation involving a missing value is that the computed variable has a missing value as well—in this case 72,890 times or nearly 41% of the records in the claims file. If one were to later sum the net charges for all of the claims for a given customer, for example, the result would be off by as much as the sum of the total_charges from records with a missing copay value.

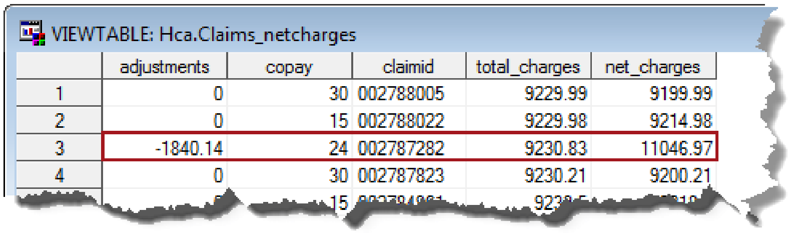


	copay	claimid	total_charges	net_charges
1	30	002788005	9229.99	9199.99
2	15	002788022	9229.98	9214.98
3	24	002787282	9230.83	9206.83
4	30	002787823	9230.21	9200.21
5	15	002784961	9233.5	9218.5
6	.	002785748	9232.[illegible]9	.
7	35	002785750	9232.59	9197.59
8	24.25	002783939	9234.67	9210.42
9	20	002782111	9236.77	9216.77
10	.	002782996	9235.[illegible]5	.
11	20	002783160	9235.57	9215.57
12	20	002783220	9235.5	9215.5

At this point, your planned data transformations require a decision to be made (and documented) about how the missing copay values will be handled. If, after checking with the subject matter experts and your project stake holders you come to a conclusion that missing values of "copay" do, in fact, represent a copay amount of "0", the best thing to do is set the missing values to 0 as one of the first DATA steps in your program. Again, documenting the decision to transform this variable programmatically is important in that your data product is likely an input to another analytic or reporting process and without this documentation, the validation of downstream analyses against the source system data will be overly complicated.

In the previous PROC MEANS step, you also notice that the value of "adjustments" does not range from 0 to a positive number but is, rather, always a negative number. Again, it is important to document this deviation from the original data dictionary specification so that a more sophisticated calculation of net_charges (wherein both adjustments and copayments are substracted from total_charges) does not result in actually "adding" the amount of the adjustments to the gross charges instead of subtracting it.

```
DATA hca.claims_netcharges;
 SET hca.claims;
 IF copay = . THEN copay=0;
 IF adjustments = . THEN adjustments = 0;

 net_charges=total_charges - copay - adjustments;

 KEEP CLAIMID NET_CHARGES COPAY TOTAL_CHARGES adjustments;
RUN;
```

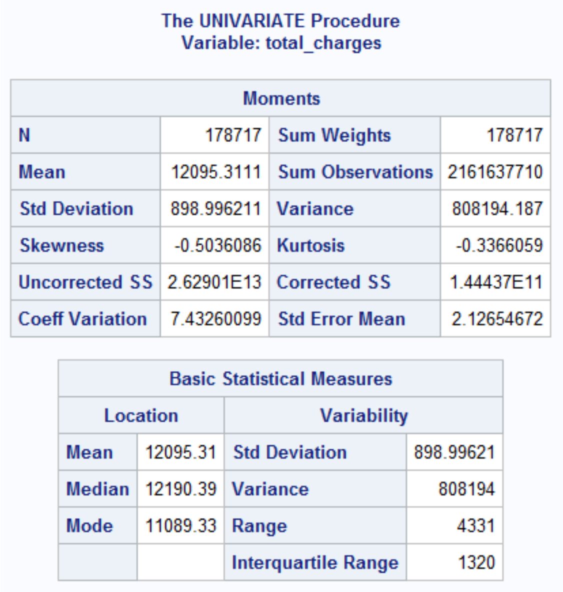


	adjustments	copay	claimid	total_charges	net_charges
1	0	30	002788005	9229.99	9199.99
2	0	15	002788022	9229.98	9214.98
3	-1840.14	24	002787282	9230.83	11046.97
4	0	30	002787823	9230.21	9200.21

Again, the solution to this discrepancy between the data dictionary and the dataset seems self-evident (and it probably is), but any time you are making an assumption about the data rather than relying on the written documentation, you should confirm your assumptions about the data and make sure the assumptions you are making are documented. You will also notice that in recoding the missing values to "0", the "Missing Values" note is no longer generated to the log. A good rule of thumb regarding calculations made in the DATA step is that they should be coded using appropriate logic such that you do not receive NOTES, WARNINGS, or ERRORS regarding missing values or implicit data conversions. This guidance is discussed again later in the paper, but beyond keeping your log file uncluttered, it also serves to provide solid evidence that you truly understand your data and are making conscious decisions that control the outcome of your programs—not simply accepting implicit conversions that may threaten the quality of your data product.

As the MEAN value of a measurement can be somewhat misleading when trying to understand all values in a dataset, you might also want to use PROC UNIVARIATE to get a fuller picture of the distribution of values within a given variable. In the following example, you see, again, the total number of records in the dataset is 178,717, but in addition to the mean and standard deviation, there are additional measures that can be useful in understanding the centrality, shape, and variability of the variable's distribution (especially useful are the median and skewness).

```
PROC UNIVARIATE DATA=hca.claims;
 VAR total_charges ;
RUN;
```

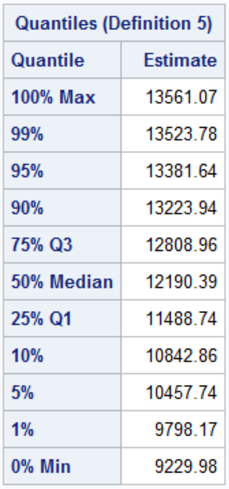


The UNIVARIATE Procedure
Variable: total_charges

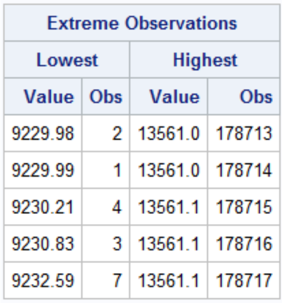
Moments			
N	178717	Sum Weights	178717
Mean	12095.3111	Sum Observations	2161637710
Std Deviation	898.996211	Variance	808194.187
Skewness	-0.5036086	Kurtosis	-0.3366059
Uncorrected SS	2.62901E13	Corrected SS	1.44437E11
Coeff Variation	7.43260099	Std Error Mean	2.12654672

Basic Statistical Measures			
Location		Variability	
Mean	12095.31	Std Deviation	898.99621
Median	12190.39	Variance	808194
Mode	11089.33	Range	4331
		Interquartile Range	1320

In addition to these measures, PROC UNIVARIATE provides quantile values and lists of the extreme observations that can also be useful in providing insight as to whether the values of a particular measure are behaving as you expect them to. For example, in the "claims" dataset, it looks like there is a roughly $4200 spread between the lowest and highest values of "total_charges". If this dataset is meant to contain all claims from a given time-period (across all types of healthcare encounters from annual check-ups to lung transplants) there is probably something wrong with the dataset, but if it is reasonable that the range of values for this particular subset of claims is $4200, then there is no cause for alarm.



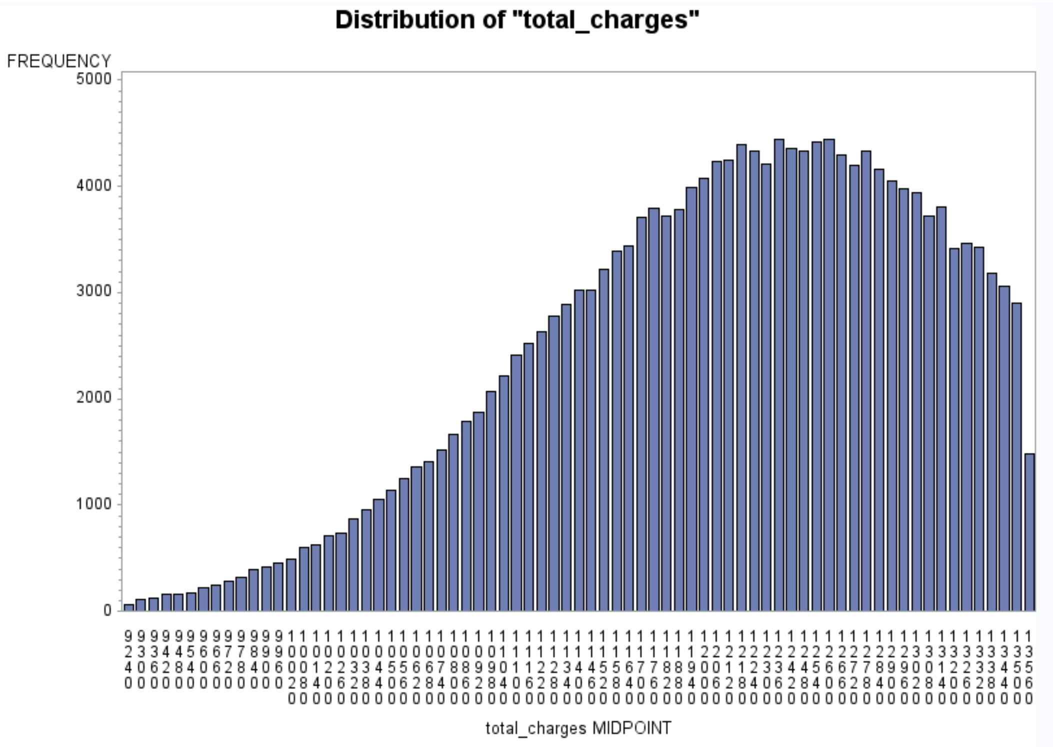
Quantiles (Definition 5)	
Quantile	Estimate
100% Max	13561.07
99%	13523.78
95%	13381.64
90%	13223.94
75% Q3	12808.96
50% Median	12190.39
25% Q1	11488.74
10%	10842.86
5%	10457.74
1%	9798.17
0% Min	9229.98



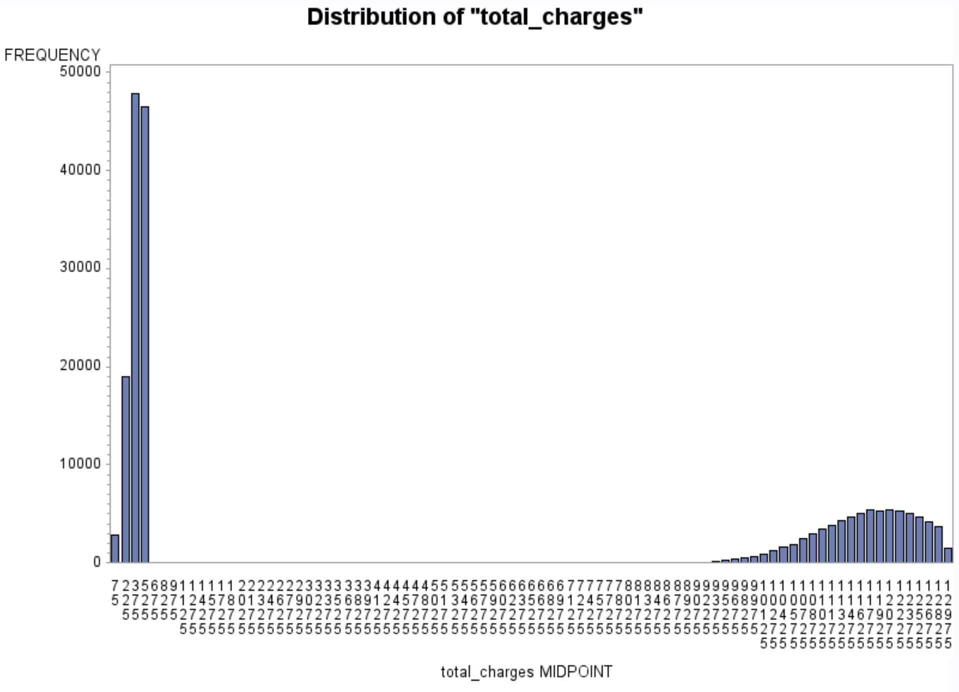
Extreme Observations			
Lowest		Highest	
Value	Obs	Value	Obs
9229.98	2	13561.0	178713
9229.99	1	13561.0	178714
9230.21	4	13561.1	178715
9230.83	3	13561.1	178716
9232.59	7	13561.1	178717

Finally, it might also be a good idea to visually inspect the values by doing a simple graph of the distribution. This technique can reveal some patterns in the data that elude all but the most skilled statistician or data manager looking only at the statistics related to the distribution. In the following example, the frequency of the "total_charges" values found in the dataset are rendered in a bar chart using PROC GCHART. Again, the question becomes one that needs to be informed by some understanding of the variable you are studying; is there any reason to question whether these data accurately reflect the processes that gave rise to them?

```
TITLE 'Distribution of "total_charges"';
PROC GCHART DATA=hca.claims;
    VBAR
     total_charges /
    TYPE=FREQ;
RUN;
QUIT;
```



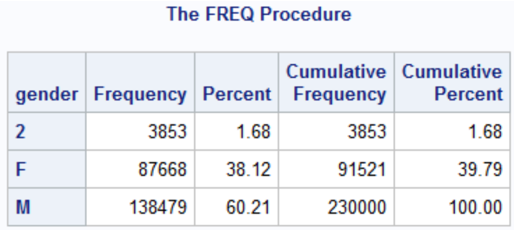
Assuming that this dataset reflects some subset of claims in which it is reasonable to believe that the charges incurred are in this roughly $4000 range, there is no need to further interrogate these data based on the review of this variable alone. If on the other hand you were studying the same subset of claims and encountered the following bimodal distribution in which a very high number of the claims are in the range of $75 - $500, but in which there are also still claims in the $13,000 area, you might need to study the dataset further.



Do all of the records in the "spike" on the left share some characteristic that the other records do not possess? Can you create a distribution that seems more reasonable by deleting all records with a given characteristic—a similar category of healthcare coverages or a specific group plan id value? Were these all records that were processed after a recent software upgrade to the claims generation software? The point here is that beyond doing a full-scale process and software validation, there is no formula for definitively concluding that something is "wrong" with any given variable that has values within the range specified in the data dictionary. However, one of the responsibilities of a data manager is to become familiar enough with the data and the business processes underlying them that you can either explain these types of patterns in the data or recognize that they do not belong. As will be discussed later in the paper, you can build some very powerful tools to help you assess data quality for ongoing, repeated data management processes, but you first need to understand the data and the relationships between various combinations of variables. You need to develop an appreciation for the business processes that give rise to your data so that you can develop a reasonable understanding for how specific data points "should" behave. It takes time to develop this knowledge and you will need to engage individuals throughout your organization to understand what it is they do, how they do it, and how those processes interact with others within the organization, but exercising your curiosity in this way will make you that much more valuable in your data management role.

Categorical Variable Confirmation (String or Numeric). – Compared to profiling numeric measurements, profiling numeric and string classification variables seems relatively simple. As was the case with numeric measurements, the first step in this process is to confirm the data dictionary description of the allowable values. In the following example, PROC FREQ is used to confirm that the values found in the "gender" variable in our healthcare plan's "membership" file are limited to "M" and "F".

```
PROC FREQ DATA=hca.members;
 TABLE gender ;
RUN;
```



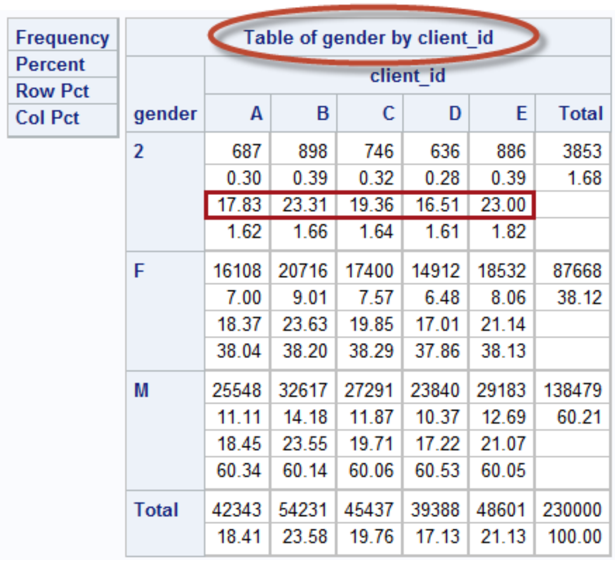
The FREQ Procedure

gender	Frequency	Percent	Cumulative Frequency	Cumulative Percent
2	3853	1.68	3853	1.68
F	87668	38.12	91521	39.79
M	138479	60.21	230000	100.00

Unfortunately in this case, in addition to "M" and "F", the value "2" makes an appearance in roughly 3800 (1.68%) of the records in this dataset. Given the relatively small number of records afflicted with this error, you might be tempted to just let it slide—reasoning that "it probably won't significantly impact analysis if we just change those to missing". The reason you should not simply ignore even small anomalies like this is precisely because you do not know what caused them. Perhaps the error occurred because the charges incurred on these claims were so high that they triggered a manual re-entry of the data which also (consequently) resulted in the "gender" value being miscoded. Simply dropping the record or blithely assigning a missing value to gender for all of these records may very well have a significant impact on reporting and analysis unless you can at least show that it was not due to a systematic error that corresponds to other characteristics of the record. To perform your due diligence in this case, you might begin researching this error by looking at the cross-tab frequency distribution of gender with other categorical variables in the dataset that can begin to help you determine the root cause of the problem. In this case, you could look at the values of gender in the context of different client companies, coverage plans, etc.

```
PROC FREQ DATA=hca.members;
 TABLE gender*client_id gender*coverage;
RUN;
```

In the first result table from this PROC step, you see that the offending value is equally distributed across the five client companies represented in the data set, so it is unlikely that the assignment of the erroneous value "2" is related to a specific client account.

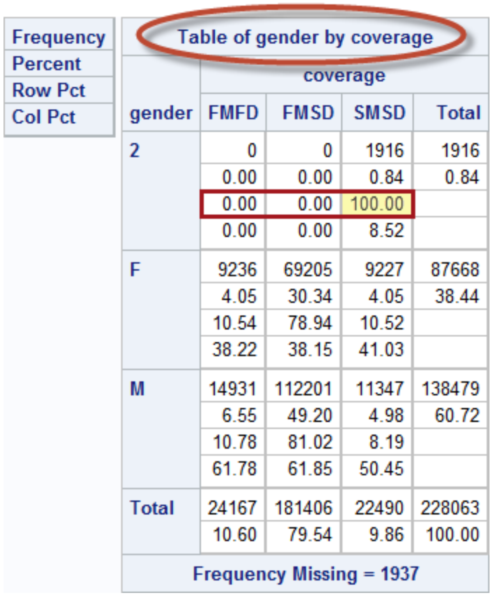


Frequency
Percent
Row Pct
Col Pct

Table of gender by client_id

gender	client_id A	B	C	D	E	Total
2	687	898	746	636	886	3853
	0.30	0.39	0.32	0.28	0.39	1.68
	17.83	23.31	19.36	16.51	23.00	
	1.62	1.66	1.64	1.61	1.82	
F	16108	20716	17400	14912	18532	87668
	7.00	9.01	7.57	6.48	8.06	38.12
	18.37	23.63	19.85	17.01	21.14	
	38.04	38.20	38.29	37.86	38.13	
M	25548	32617	27291	23840	29183	138479
	11.11	14.18	11.87	10.37	12.69	60.21
	18.45	23.55	19.71	17.22	21.07	
	60.34	60.14	60.06	60.53	60.05	
Total	42343	54231	45437	39388	48601	230000
	18.41	23.58	19.76	17.13	21.13	100.00

However, the second cross-tab table provides additional information about the origins of these erroneous values. It seems they are limited to the SMSD (subscriber medical, subscriber dental) coverage plan.



Frequency
Percent
Row Pct
Col Pct

Table of gender by coverage

gender	coverage FMFD	FMSD	SMSD	Total
2	0	0	1916	1916
	0.00	0.00	0.84	0.84
	0.00	0.00	100.00	
	0.00	0.00	8.52	
F	9236	69205	9227	87668
	4.05	30.34	4.05	38.44
	10.54	78.94	10.52	
	38.22	38.15	41.03	
M	14931	112201	11347	138479
	6.55	49.20	4.98	60.72
	10.78	81.02	8.19	
	61.78	61.85	50.45	
Total	24167	181406	22490	228063
	10.60	79.54	9.86	100.00

Frequency Missing = 1937

Of course, this doesn't mean that assigning individual health plan members to this plan "caused" the erroneous value to be assigned (note that there are "F" and "M" values identified under this coverage type as well), but this result suggests that you should focus your research on this coverage plan—and, perhaps, recent changes to it. At the very least, when you alert the source system administrator about this error, you will be helping them a great deal if you can describe some of the trouble-shooting steps you have taken so far. Perhaps there was a recent maintenance release of a software package that touches only the SMSD claims or maybe there was a recent product trial that was rolled out only to that group of patients. The point, again, is to think about the possible relationships between this value and other characteristics of the records containing the error, test those hypotheses, and follow where the data take you. One additional piece of information about this pattern of erroneous value codes is that there were also a number of missing values generated in this crosstab; those will be addressed by the analysis presented in the next example.

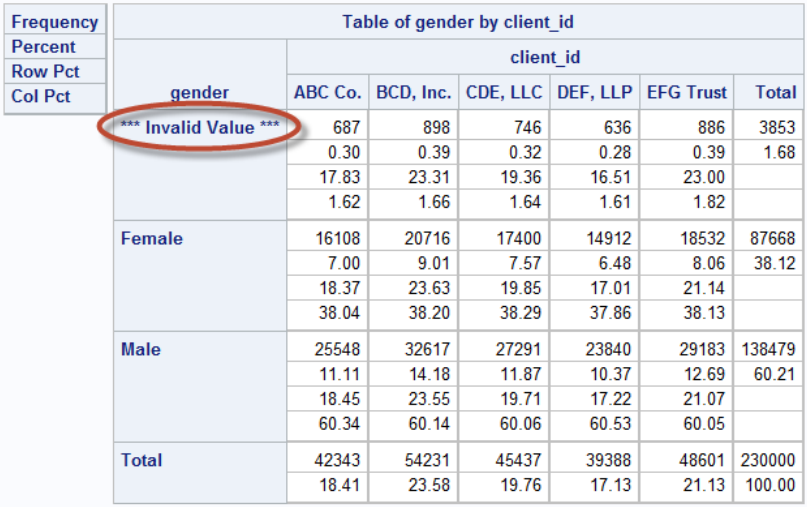
This PROC FREQ approach to the analysis of individual categorical variables is fine when there are relatively few variables in your dataset and you have a relatively small number of valid category variables, but beyond a few variables with a few categories each, you need a more robust, formalized approach to the analysis. One way to perform a more thorough check of a large number of variables and categories is to create user-defined formats and assign the formats to the variables being profiled (Cody, 1999).

As described in-depth elsewhere (Cody, 2010; Scerbo, 2007) FORMATS are instruction sets that tell SAS how to <u>represent</u> data stored in a variable. For example, applying the SAS format MMDDYY10. changes the representation of a SAS date variable to the familiar format "mm/dd/yyyy"—for example, 17850 becomes "11/14/2008". INFORMATS, conversely, are instruction sets that determine how values should be interpreted as they are read into a variable. User-defined formats (and informats) are simply those formats for which you define your own instructions (e.g., 0 = 'No' / 1 = 'Yes'). As shown in the following example, you can use PROC FORMAT to create user-defined formats that assign to your data the values described in a data dictionary. In addition to the coded values specified by the data dictionary, however, these formats can also include categories for missing values (' ') and "OTHER" in order to identify both records with missing values and those with values other than those specified by the data dictionary.

```
PROC FORMAT;
 VALUE $gender 'F' = 'Female'
               'M' = 'Male'
               ' ' = 'Missing'
             OTHER = '+++ Invalid Value ***';
 VALUE $clients 'A' = 'ABC Co.'   'B' = 'BCD, Inc.'   'C' = 'CDE, LLC'
                'D' = 'DEF, LLP'  'E' = 'EFG Trust'
                ' ' = 'Missing'
              OTHER = '*** Invalid Value ***';
 VALUE $plan_name 'SMSD' = 'Subscriber Medical Subscriber Dental'
                  'FMFD' = 'Family Medical Family Dental'
                  'FMSD' = 'Family Medical Subscriber Dental'
                     ' ' = 'Missing'
                   OTHER = '*** Invalid Value ***';
RUN;
```

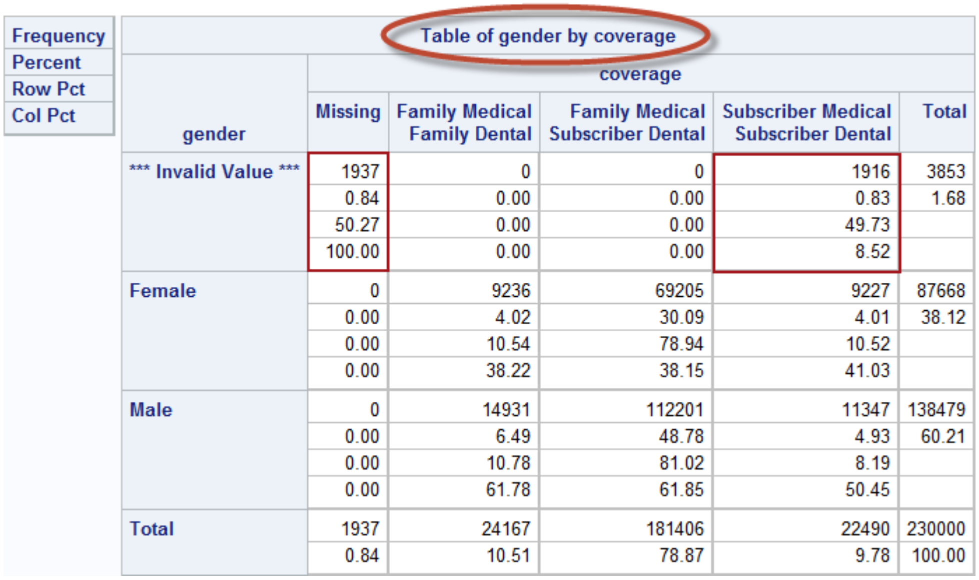
With these formats in place, it is much easier to identify undocumented category values in a large number of variables with several valid response categories and to identify the pattern of occurrence of both missing and invalid values. As shown below, the invalid values stand out very clearly in the PROC FREQ output. Please note, however, that using this technique, <u>all</u> invalid values are lumped into one category ("Invalid Value") and therefore, if you discover invalid values, you should rerun the FREQUENCY procedure again without the formats to identify the individual invalid values.

```
PROC FREQ DATA=hca.members;
 FORMAT gender $gender. client_id $clients. coverage $plan_name.;
  TABLE gender*client_id  gender*coverage /MISSING;
RUN;
```



Frequency Percent Row Pct Col Pct	Table of gender by client_id						
		client_id					
	gender	ABC Co.	BCD, Inc.	CDE, LLC	DEF, LLP	EFG Trust	Total
	*** Invalid Value ***	687	898	746	636	886	3853
		0.30	0.39	0.32	0.28	0.39	1.68
		17.83	23.31	19.36	16.51	23.00	
		1.62	1.66	1.64	1.61	1.82	
	Female	16108	20716	17400	14912	18532	87668
		7.00	9.01	7.57	6.48	8.06	38.12
		18.37	23.63	19.85	17.01	21.14	
		38.04	38.20	38.29	37.86	38.13	
	Male	25548	32617	27291	23840	29183	138479
		11.11	14.18	11.87	10.37	12.69	60.21
		18.45	23.55	19.71	17.22	21.07	
		60.34	60.14	60.06	60.53	60.05	
	Total	42343	54231	45437	39388	48601	230000
		18.41	23.58	19.76	17.13	21.13	100.00

Missing values are able to be processed for inclusion in the crosstab tables because the "/MISSING" switch was included in the TABLE statement. If the "Missing Value" entries had not been included in the user-defined formats, these values would still have been represented in the table but would have been assigned to the "OTHER" category and been tagged as an "*** Invalid Value ***".



Frequency
Percent
Row Pct
Col Pct

Table of gender by coverage

gender	coverage: Missing	Family Medical Family Dental	Family Medical Subscriber Dental	Subscriber Medical Subscriber Dental	Total
*** Invalid Value ***	1937	0	0	1916	3853
	0.84	0.00	0.00	0.83	1.68
	50.27	0.00	0.00	49.73	
	100.00	0.00	0.00	8.52	
Female	0	9236	69205	9227	87668
	0.00	4.02	30.09	4.01	38.12
	0.00	10.54	78.94	10.52	
	0.00	38.22	38.15	41.03	
Male	0	14931	112201	11347	138479
	0.00	6.49	48.78	4.93	60.21
	0.00	10.78	81.02	8.19	
	0.00	61.78	61.85	50.45	
Total	1937	24167	181406	22490	230000
	0.84	10.51	78.87	9.78	100.00

With the missing values included in the cross-tab table, you can see that the invalid gender values are split about evenly between those claim records with a missing value for "coverage" and the coverage value "SMSD". The cause of the missing values for "coverage" (which you will also discover in your analysis of that variable) are likely related to the invalid "gender" values because none of the records with a valid "gender" value are missing their "coverage" value. It may ultimately turn out to be the case that these two errors have completely independent causes, but the data, at this point, suggest they have something in common. In either case, now you can present your issue to the administrator of the source system with some hard facts to pursue and not simply complain to them that "your data are wrong".

Of course, if the list of acceptable values for a given variable is large (e.g., code label pairs from a large standardized taxonomy such as diagnosis related group (DRG) codes in medical claims), it becomes a bit impractical to manually create formats as in the previous example. If you have a dataset of these code/label pairs, you can save considerable time and effort by creating your format programmatically. In the following example, a user-defined format for DRG codes is produced based on source data downloaded from the Centers for Medicare and Medicaid Services.

When producing a data-driven, user-defined format, PROC FORMAT requires that the data being used in the creation of the format have a specific structure. In this example, the variables "start", "label", and "fmtname" are written to the dataset "work.drg" which will serve as the input (control) file to PROC FORMAT. Because the descriptive labels for the numeric DRG codes can be somewhat lengthy for the purposes of profiling the dataset, instead of using the DRG description as the "label" for the format, a zero-filled representation of the numeric code will be written to the "label" variable in the work.drg dataset. The static value "drgcodes" will ultimately serve as the format name.

```
DATA work.drg;
 SET lookups.drg_codes END=last;
     start=drg_code;
     label=PUT(drg_code,Z3.);
     fmtname='drgcodes';
 OUTPUT;
```

If the DATA step that produces the "drg" dataset stopped here, the format created from it would contain only the code/label pairs for valid values of "drg", but the format also needs to be able to process "OTHER" values as well as missing values. Therefore, the DATA step will write out two additional records beyond those read from the "lookups.drg_codes" dataset. The first of these additional records assigns "O" (for "other") as the value for the variable "hlo". PROC FORMAT will recognize this record as the source of a label for "Other" values. The second additional record written to dataset "work.drg" has a null value for "hlo" and will be recognized by PROC FORMAT as providing the label for missing values.

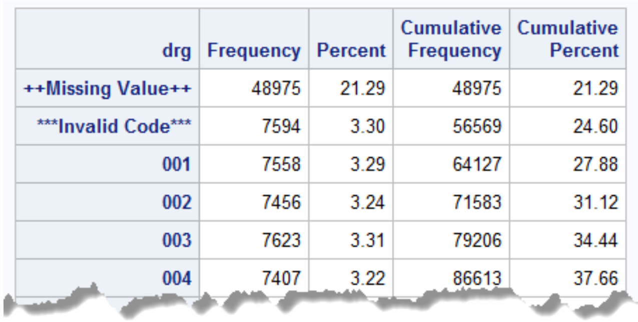
```
IF last THEN DO;
        hlo='o';
      start= .;
      label= '***Invalid Code***';
     OUTPUT;
       start= .;
         hlo='';
       label= '++Missing Value++';
     OUTPUT;
  END;
KEEP start label fmtname hlo;
RUN;
```

With the control file created, PROC FORMAT is executed with the value of "CNTLIN" (input control file, or, control in) specified as the "work.drg" dataset and the result is creation of the "drgcodes" format being written to the WORK library.

```
PROC FORMAT CNTLIN=work.drg LIBRARY = WORK;
RUN;
```

This format, in turn, can be used like the manually written formats created in the previous examples to analyze the frequency distribution of the claims dataset. With this format applied to our ficticious healthcare claims dataset, you see that 21% of the records contain a missing value and 3.3% contain an invalid drg compared to our drg lookup dataset.

```
PROC FREQ DATA=hca.claims;
 FORMAT drg drgcodes.
  TABLE drg /MISSING;
RUN;
```



drg	Frequency	Percent	Cumulative Frequency	Cumulative Percent
++Missing Value++	48975	21.29	48975	21.29
Invalid Code	7594	3.30	56569	24.60
001	7558	3.29	64127	27.88
002	7456	3.24	71583	31.12
003	7623	3.31	79206	34.44
004	7407	3.22	86613	37.66

Once you are sure that the individual variables contain acceptable values, there are likely conditional logical checks of your data that you will want to perform using IF/THEN statements to make sure the data conform to the business rules associated with a process. Many of these will not become apparent to you until you start transforming the data and producing initial analytic results, but some are more self-evident and making sure your data conforms to at least the basic business rules can help you avoid more complex data manipulations later in your program. The most important of these checks is actually a step that you will probably want to perform even before evaluation of individual variables—checking for duplicate records.

CHECKING FOR DUPLICATE RECORDS. - In addition to the data dictionary's information about the individual data elements, one of the first things you should learn about a dataset is the combination of columns (variables) that uniquely identify an individual record. Often, this will be some sort of system-generated internal record identification number. For example, a health plan's membership file might have a system-generated "subscriber_id" and you want to make sure that you don't have any records with duplicate "subscriber_id" values. One quick way to identify any duplicates in this file would be to use the PROC SQL COUNT function with a HAVING clause to create a table of records with duplicate subscriber_ids. The following query counts the number of records with a given subscriber_id and outputs the subscriber_id and the number of records associated with that subscriber_id in cases where the subscriber_id is found to be associated with more than 1 record.

```
PROC SQL;
 CREATE TABLE hca.duplicate_subscribers AS
   SELECT subscriber_id,count('x') AS num_dups
     FROM hca.members
  GROUP BY subscriber_id
    HAVING num_dups > 1;
QUIT;
```

Of course, if there are no duplicates (as there should not be, according to the subject matter expert in charge of the membership file), you will get a log file message like the following:

```
740  PROC SQL;
741   CREATE TABLE hca.duplicate_subscribers AS
742     SELECT subscriber_id,count('x') AS num_dups
743       FROM hca.members
744    GROUP BY subscriber_id
745      HAVING num_dups > 1;

NOTE: Table HCA.DUPLICATE_SUBSCRIBERS created, with 0 rows and 2 columns.
```

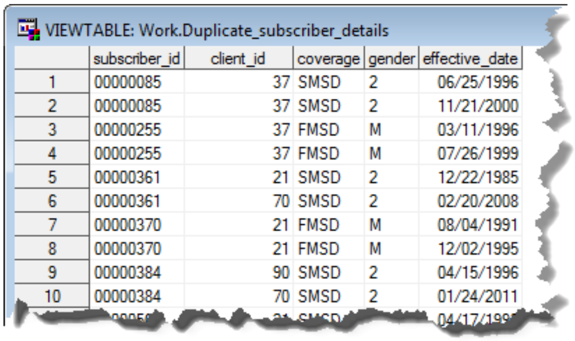
If, on the other hand, there are multiple records that share a subscriber_id, those unique subscriber_ids will be output to the dataset "hca.duplicate_subscribers". In the following example, there were 3789 subscriber_id values that were found on more than one member record.

```
NOTE: Table HCA.DUPLICATE_SUBSCRIBERS created, with 3789 rows and 2 columns.
```

Not that you are happy with this finding, but this is exactly the type of thing you are looking for—the purpose for profiling your data in the first place. In order to study these records more closely, you need to create a dataset comprised of the duplicate records from the members dataset. This is done using PROC SQL to create a dataset of the records in the membership file that have a subscriber_id in the dataset of duplicate subscriber_ids.

```
PROC SQL;
 CREATE TABLE duplicate_subscriber_details AS
   SELECT *
     FROM hca.members
    WHERE subscriber_id IN (SELECT subscriber_id
                                FROM hca.duplicate_members)
 ORDER BY subscriber_id;
QUIT;
```

Sometimes a visual inspection of a "duplicates" dataset will shed light on the situation. In this case, you see that each of these duplicates seems to differ in terms of their "effective_date". Some also differ in terms of the "client_id", but some do not (e.g., subscriber_id "00000085"). This is the point, too, at which a bit of subject matter knowledge can be applied to the data. In this case, the "client_id" is defined as the client company providing the coverage to their employees, and the effective date is the date at which coverage was begun. The "subscriber" could have left his/her job with client 37 and come back to work there four years later—or, dropped coverage and then picked it up again due to a life-event. The point here is that you came into this data profiling exercise believing that records in the member dataset were uniquely identifiable by the subscriber_id. Based on the previous PROC SQL steps you learned this was not correct, but now you have a theory about what makes records in this dataset unique—the combination of "subscriber_id" and "effective_date".



VIEWTABLE: Work.Duplicate_subscriber_details

	subscriber_id	client_id	coverage	gender	effective_date
1	00000085	37	SMSD	2	06/25/1996
2	00000085	37	SMSD	2	11/21/2000
3	00000255	37	FMSD	M	03/11/1996
4	00000255	37	FMSD	M	07/26/1999
5	00000361	21	SMSD	2	12/22/1985
6	00000361	70	SMSD	2	02/20/2008
7	00000370	21	FMSD	M	08/04/1991
8	00000370	21	FMSD	M	12/02/1995
9	00000384	90	SMSD	2	04/15/1996
10	00000384	70	SMSD	2	01/24/2011

Modifying the previous query used to identify duplicate records, you can test this new theory by adding "effective_date" to both the SELECT statement and GROUP BY clause.

```
PROC SQL;
 CREATE TABLE hca.duplicate_subscribers AS
   SELECT subscriber_id, effective_date, count('x') AS num_dups
     FROM hca.members
  GROUP BY subscriber_id, effective_date
    HAVING num_dups > 1;
QUIT;
```

Now you see from the SAS log that you correctly understand what makes the records in the members table unique—the combination of subscriber_id and effective-date.

```
NOTE: Table HCA.DUPLICATE_SUBSCRIBERS created, with 0 rows and 2 columns.
```

Admittedly, discovering the genesis of duplicate records is not always this easy. Often, the combination of factors that led to duplicates is much more subtle. For example, we might have found that all of the duplicates were part of a particular category of record (e.g., all had the same "coverage") but that the error only showed up when the effective date was in a particular date range (e.g., June and July of 2005). It is important that you discover the algorithm used to describe these types of errors because these are precisely the type of systematic errors imposed by the systems used to process and manage data that can significantly distort your view of the processes you are studying.

Another lesson to be learned from this example is that as a data manager you should trust no one regarding the veracity of the data. That sounds incredibly cynical, but let the data speak for itself; interrogate the data and challenge accepted notions about the nature of the data. Even data stewards with deep subject matter knowledge may, from time to time, have an incorrect conceptual model of the data in their systems. Actually, this happens much more often than you probably think it does. Regardless of the reason this information was incorrectly communicated to you, make sure that you both update the data dictionary or other appropriate meta-data source to correct this description of the data and communicate it to the data steward tasked with managing the source data. The majority of times when you communicate a finding like this to a data steward their response is something along the lines of "oh yeah, I guess that is right…sorry". They likely understand the process that gave rise to the data very well (i.e., that a healthplan member can have more than one effective date) but may not think about the implications of that operational fact on downstream analytic processes. Regardless of the reason for the miscommunication, the important thing is that the documentation gets corrected so the next new person dealing with these data does not have to discover this situation on their own.

HARD-CODED CORRECTIONS

Upon identifying anomalies in your data profiling, any suspected errors you find should be brought to the attention of the source system administrator and either corrected at the source or documented as not actually representing errors. For example, if the earlier "gender=2" issue was, in fact, caused by an upstream coding error, the error should be corrected and the records reprocessed in the source system. That way, operational systems will contain the correct data, other downstream processes that depend on those data will not run into the same problem you encountered, and no additional corrections need to be made to your dataset. The alternative is to hard-code these corrections as part of your analytic program—introducing further potential points of failure in the resulting dataset. Such hard-coded alterations to the data should be avoided if at all possible because they add unnecessary complexity to the analytic program, require additional documentation, and may actually introduce new errors to the data after the source of the anomaly has been corrected in the upstream data source.

If forced to make hard-coded corrections to individual records (or sub-sets of records), make sure you document the change and maintain an active dialogue with the source system owner so that you are aware of the status of any pending "fix" to the solution. As an example, perhaps the source system administrator was able to give you the correct gender classifications for the "gender=2" members in the previous examples—but was, for whatever reason, not able to reprocess the data to give you a corrected data file. You might hard code the correction as follows from the dataset "corrected_gender" that contains the "memberid" and the corrected value of "gender".

```
/*2012-07-13
  Temporary fix to correct "gender" anaomalie caused by system
  upgrade to SMSD processing module*/
PROC SQL;
 UPDATE hca.members a
    SET gender = 'F'
  WHERE memberid IN (SELECT memberid b
                       FROM corrected_gender
                                 WHERE b.gender = 'F');
 UPDATE hca.members a
    SET gender = 'M'
  WHERE memberid IN (SELECT memberid b
                       FROM corrected_gender
                                 WHERE b.gender = 'M');
QUIT;
```

Again, this type of hard-coding to correct data errors should only be used as a last resort, as it introduces potential vulnerabilities to the reliability and validity of the analytic dataset. Suppose the data are corrected in the source system prior to next month's generation of the source dataset, but in that process additional corrections are made to the membership file—rendering your "corrected_gender" dataset incorrect for one or more members. Now the fix you implemented is introducing new errors that become harder and harder to find and correct. Communication with the source system administrator is crucial to make sure you know when your code should be altered again to remove the hard-coded fix. You should also ask yourself how important this correction was in the first place. Do any of the analyses rely on gender? What are the consequences of not having these data available for this month's reporting cycle? All of these factors should be considered before throwing in hard-coded corrections to data that otherwise would come from an operational system.

Regardless of how corrections to the data are achieved, however, once you have a dataset that accurately reflects the data in the operational system you can begin the work of transforming it for use in analysis and reporting. It is often surprising how many changes need to be made in order to analyze data from an operational system. One might think that the data from the operational system are all one needs in order to study the processes supported by those systems. However, operational systems often store data in a format that is not conducive to analysis and variables may need to be parsed and/or combined to create analytic categories, or the data may need to be rolled-up or expanded to achieve the appropriate granularity for analysis. Commonly encountered transformations of these types are considered in the following section.

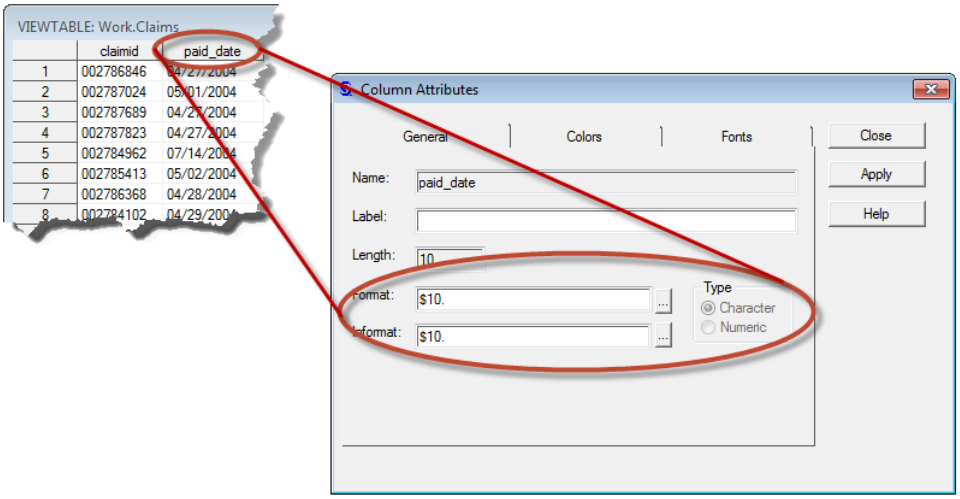
DATA TRANSFORMATIONS

Once you feel comfortable that the data are clean at the level of the individual records and variables and you understand the granularity of the data with which you are working, it is time to start planning the other data transformations necessary to produce the analytic dataset you intend to deliver. In reality, some of the data "cleaning" will actually happen in an iterative fashion as you begin performing transformations and discover additional, subtle ways in which the data do not accurately reflect the business processes from which they arose. However, some of tasks that people refer to as data cleaning do not result in substantive changes to the data elements, but simply reflect transformations of the data that are necessary in order to make your data more useful (or more user-friendly) for reporting and analysis. Regardless of the reason the transformations are made, care should be taken in documenting them. In addition, such transformations should be clearly communicated to subject matter experts and project stakeholders to ensure that everyone is in agreement with the necessity for the transformation, the assumptions underlying it, and the specific manner in which it will be coded.

DATA TYPE CONVERSIONS

One of the most common conversions one needs to make in producing an analytic dataset is converting STRING variables to NUMERIC variables and vice versa. These conversions involve the INPUT and PUT functions.

The INPUT function tells SAS how to "read" data into a variable using INFORMATS. In the following example, the dataset "claims" contains a string variable "paid_date". Although it "looks" like a date, you would not be able to calculate, for example, the number of days between the bill_date and the paid_date because the latter is stored as a string variable—not a numeric date.


VIEWTABLE: Work.Claims

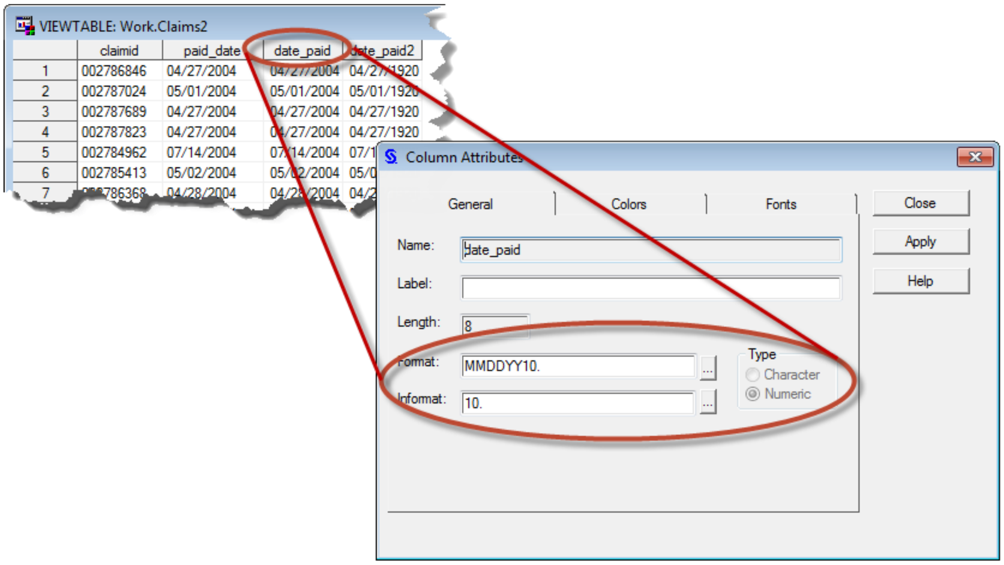
	claimid	paid_date
1	002786846	04/27/2004
2	002787024	05/01/2004
3	002787689	04/27/2004
4	002787823	04/27/2004
5	002784962	07/14/2004
6	002785413	05/02/2004
7	002786368	04/28/2004
8	002784102	04/29/2004

Column Attributes

Name: paid_date
Label:
Length: 10
Format: $10.
Informat: $10.
Type
● Character
○ Numeric

In order to convert "paid_date" to a numeric value, the INPUT function is used in the following DATA step to create a new variable "date_paid" that has a numeric data type. Specifically, this INPUT function specifies that the data should be read into the new "date_paid" variable using the MMDDYY10. INFORMAT (or, instruction set). This INFORMAT tells SAS that in converting the string to a numeric value, the two characters prior to the separator "/" represent the month, the next two characters (after the first separator) represent the day and the last four characters represent the year—MMDDYY. The "10" in MMDDYY10. specifies the total length of the string being read. If, as in the case of "date_paid2" you were to specify MMDDYY8. as the INFORMAT, you would end up with a date that does not accurately reflect the "paid_date" because the MMDDYY8. INFORMAT instructs SAS to only read the first eight characters of the "paid_date" variable in determining the date.

```
DATA claims;
 SET claims;
 FORMAT DATE_PAID DATE_PAID2 MMDDYY10.
  date_paid = INPUT(paid_date,MMDDYY10.);
  date_paid2 = INPUT(paid_date,MMDDYY8.);
RUN;
```


VIEWTABLE: Work.Claims2

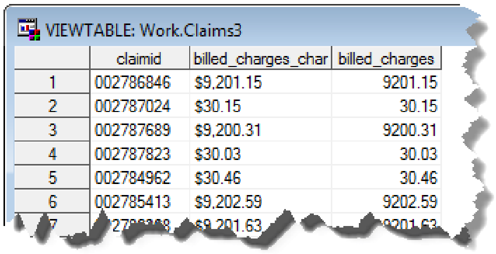
	claimid	paid_date	date_paid	date_paid2
1	002786846	04/27/2004	04/27/2004	04/27/1920
2	002787024	05/01/2004	05/01/2004	05/01/1920
3	002787689	04/27/2004	04/27/2004	04/27/1920
4	002787823	04/27/2004	04/27/2004	04/27/1920
5	002784962	07/14/2004	07/14/2004	07/1
6	002785413	05/02/2004	05/02/2004	05/0
7	002786368	04/28/2004	04/28/2004	04/2

Column Attributes

Name: date_paid
Label:
Length: 8
Format: MMDDYY10.
Informat: 10.
Type
○ Character
● Numeric

Similarly, if you have received a dataset with string versions of variables that contain dollar amounts you will be unable to use them in arithmetic expressions unless you first convert them to numbers. In the following example, the claims dataset variable "billed_charges_char" contains character data and needs to be converted to numeric data in order to be used in calculation of other metrics associated with each claim record. Using the INPUT function, a new variable "billed_charges" is created by informing SAS that in reading the "billed_charges_char" value, a numeric value can be created by ignoring the dollar sign ($) and thousand separators (,). The result is a numeric variable "billed_charges" that reflects the contents of the string variable "billed_charges_char".

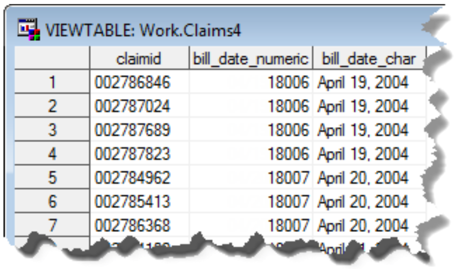
```
DATA claims3;
 SET claims3;
 billed_charges = INPUT(billed_charges_char,DOLLAR12.2);
RUN;
```



	claimid	billed_charges_char	billed_charges
1	002786846	$9,201.15	9201.15
2	002787024	$30.15	30.15
3	002787689	$9,200.31	9200.31
4	002787823	$30.03	30.03
5	002784962	$30.46	30.46
6	002785413	$9,202.59	9202.59
7	[illegible]	$9,201.63	9201.63

Conversely, if you need to write a variable out to a dataset as formatted character data and it is currently stored as numeric data, you can use the PUT function—which uses FORMATS to determine how to write the contents of a SAS variable (to a SAS dataset or other output file). In the following example, the variable "bill_date_numeric" is stored as a numeric SAS variable and you need to use it to write out the character form of the date—say, to use as an input to the process that actually prints the billing forms prior to mailing. The PUT function is used in this case to operate against the numeric date field and write out the "MonthName Day, Year" form of the corresponding date.

```
DATA claims4;
 SET hca.claims;
     bill_date_char=PUT(bill_date_numeric,WORDDATE.);
KEEP CLAIMID BILL_DATE_NUMERIC BILL_DATE_CHAR;
RUN;
```



	claimid	bill_date_numeric	bill_date_char
1	002786846	18006	April 19, 2004
2	002787024	18006	April 19, 2004
3	002787689	18006	April 19, 2004
4	002787823	18006	April 19, 2004
5	002784962	18007	April 20, 2004
6	002785413	18007	April 20, 2004
7	002786368	18007	April 20, 2004

In addition to using PUT and INPUT with the many SAS-provided formats, user-defined FORMATS and INFORMATS can also be used to convert data for output to an analytic dataset or convert text labels from source system data into numeric values to use in performing your dataset transformations. In the following example, there are modifiers that get applied to "billed_charges" in order to calculate "total_charges" in accordance with contractual agreements with individual client companies. Because these modifiers differ for each client company there needs to be a way to apply the correct modifier to each record. These values could be assigned using hard-coded IF/THEN logic as follows:

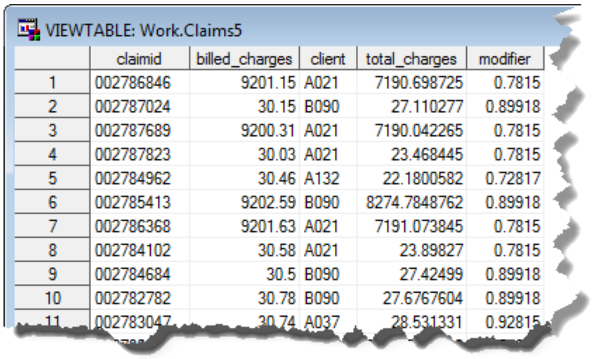
```
DATA claims5;
 SET claims;
 IF client = 'A021' then modifier = .07815;
    ELSE IF client = 'A037' THEN modifier = .92815;
    ELSE IF client = 'B090' THEN modifier = .89918;
    ELSE IF client = 'B041' THEN modifier = .65156;
    ELSE IF client = 'A132' THEN modifier = .72817;
adjusted_total = billed_charges*modifier;
RUN;
```

However, in hard-coding these values you incur a significant risk; the contractual modifiers might be changed without notification and your calculations would be in error. It would be better to dynamically assign the modifiers to your adjusted_total calculation based on the official repository of these data within your organization. Assuming you can identify this dataset and gain access to it you can considerably improve the long-term reliability of your analytic solution by creating an INFORMAT based on these data and using that INFORMAT in your calculation of "adjusted_total". Creating a user-defined INFORMAT is achieved in basically the same way as the user-defined FORMATS created previously. The one exception is that in addition to specifying "start", "label", and "hlo", you will also specify the "type" as "I" (for INFORMAT).

```
DATA work.modifiers;
 SET prod.modifiers END=last;
    Start = client;
    Label = modifier_weight;
    Type  = 'I';
    fmtname='modifiers';
 OUTPUT;
 IF last THEN DO;
        hlo='o';
      start= '';
      label= 1;
     OUTPUT;
  END;
KEEP start label type fmtname hlo;
RUN;
```

Once created, the INFORMAT can be used to dynamically value the "modifier" value by applying the INFORMAT to the "client" value using the INPUT function.

```
DATA claims5;
 SET claims;
     modifier = INPUT(client,modifiers.);
adjusted_charges = modifier*billed_charges;
RUN;
```



VIEWTABLE: Work.Claims5

	claimid	billed_charges	client	total_charges	modifier
1	002786846	9201.15	A021	7190.698725	0.7815
2	002787024	30.15	B090	27.110277	0.89918
3	002787689	9200.31	A021	7190.042265	0.7815
4	002787823	30.03	A021	23.468445	0.7815
5	002784962	30.46	A132	22.1800582	0.72817
6	002785413	9202.59	B090	8274.7848762	0.89918
7	002786368	9201.63	A021	7191.073845	0.7815
8	002784102	30.58	A021	23.89827	0.7815
9	002784684	30.5	B090	27.42499	0.89918
10	002782782	30.78	B090	27.6767604	0.89918
11	002783047	30.74	A037	28.531331	0.92815

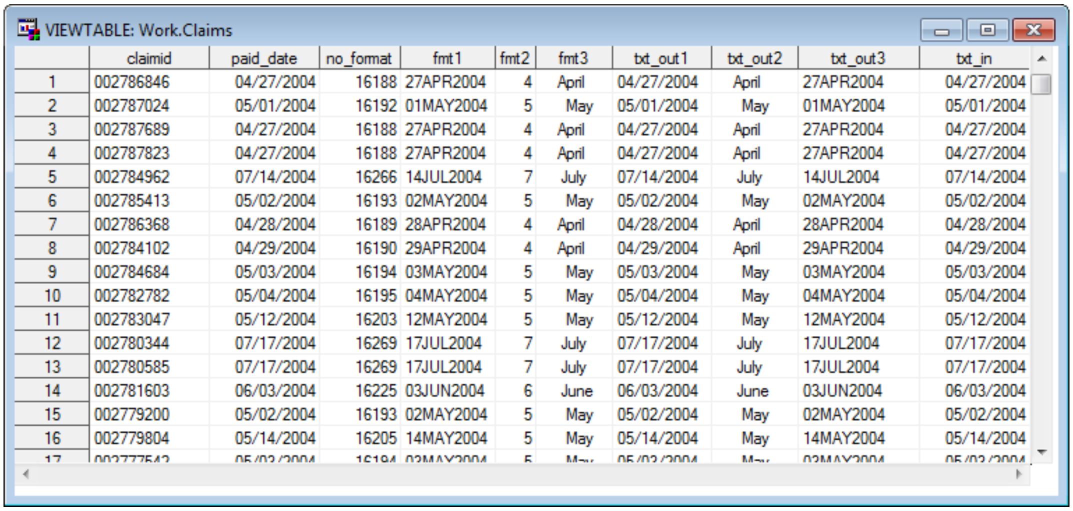
DATES, STRINGS REPRESENTING DATES, & DATE CONVERSIONS

Among the most commonly required and most misunderstood transformations performed in the creation of an analytic dataset are those associated with dates. As a reminder, it should be noted that SAS stores dates as integers indicating the number of days since January 1, 1960—e.g., 1/1/1960 is stored as the integer "0", 1/2/1960 is stored as "1", etc. Similarly, SAS stores datetime variables as the number of seconds since January 1, 1960 at midnight. The datetime 1/1/1960 at 12:01 am is stored as the integer "60". The confusion with dates and datetimes arises partially from the fact that people are used to seeing dates <u>formatted</u> in familiar ways in both SAS datasets and other electronic formats such as Microsoft Excel. When unformatted dates are presented in the dataset browser it is easy to jump to the conclusion that something is wrong—that these integers are not dates because they do not "look" like dates. Conversely, when the values of a string variable “look like” dates it can be hard to understand why they cannot be used to perform date math (e.g., number of days from date “x” to date “y”). And when the FORMATs applied to numeric date variables are changed, it is almost difficult to believe that their contents have not been altered as well.

In an attempt to help clarify all of these issues in a single example, the following DATA step presents several variants of the "paid_date" variable in the "claims" dataset. The variables "no_format", "fmt1", "fmt2", and "fmt3" contain the exact same integer value as the variable "paid_date" (which is formatted with MMDDYY10.) and differ ONLY in the format assigned to them in the FORMAT statement. This difference in formatting changes the appearance of the data they contain, but does not change, in any way, their content. When included in listings, reports, and procedure output, these variables will be presented according to their respective formats but all would yield the same result when used in arithmetic expressions.

The variables "txt_out1", "txt_out2", and “txt_out3”, on the other hand appear as if they might contain the same date value as "paid_date", but they actually contain non-numeric values produced using the PUT function to convert the “paid_date” value into a text string according to the specified FORMAT. Finally, "txt_in" contains a numeric SAS date resulting from using the INPUT function to convert “txt_out3” using the DATE9. INFORMAT.

```
DATA work.claims;
  SET hca.claims;
FORMAT fmt1 DATE9. fmt2 MONTH. fmt3 MONNAME. txt_in mmddyy10.;

          no_format = paid_date;
               fmt1 = paid_date;
               fmt2 = paid_date;
               fmt3 = paid_date;
           txt_out1 = PUT(paid_date,mmddyy10.);
           txt_out2 = PUT(paid_date,monname.);
           txt_out3 = PUT(paid_date,date9.);
             txt_in = INPUT(txt_out3,date9.);

 KEEP claimid paid_date
      no_format fmt1 fmt2 fmt3
      txt_out1 txt_out2 txt_out3 txt_in;
RUN;
```



VIEWTABLE: Work.Claims

	claimid	paid_date	no_format	fmt1	fmt2	fmt3	txt_out1	txt_out2	txt_out3	txt_in
1	002786846	04/27/2004	16188	27APR2004	4	April	04/27/2004	April	27APR2004	04/27/2004
2	002787024	05/01/2004	16192	01MAY2004	5	May	05/01/2004	May	01MAY2004	05/01/2004
3	002787689	04/27/2004	16188	27APR2004	4	April	04/27/2004	April	27APR2004	04/27/2004
4	002787823	04/27/2004	16188	27APR2004	4	April	04/27/2004	April	27APR2004	04/27/2004
5	002784962	07/14/2004	16266	14JUL2004	7	July	07/14/2004	July	14JUL2004	07/14/2004
6	002785413	05/02/2004	16193	02MAY2004	5	May	05/02/2004	May	02MAY2004	05/02/2004
7	002786368	04/28/2004	16189	28APR2004	4	April	04/28/2004	April	28APR2004	04/28/2004
8	002784102	04/29/2004	16190	29APR2004	4	April	04/29/2004	April	29APR2004	04/29/2004
9	002784684	05/03/2004	16194	03MAY2004	5	May	05/03/2004	May	03MAY2004	05/03/2004
10	002782782	05/04/2004	16195	04MAY2004	5	May	05/04/2004	May	04MAY2004	05/04/2004
11	002783047	05/12/2004	16203	12MAY2004	5	May	05/12/2004	May	12MAY2004	05/12/2004
12	002780344	07/17/2004	16269	17JUL2004	7	July	07/17/2004	July	17JUL2004	07/17/2004
13	002780585	07/17/2004	16269	17JUL2004	7	July	07/17/2004	July	17JUL2004	07/17/2004
14	002781603	06/03/2004	16225	03JUN2004	6	June	06/03/2004	June	03JUN2004	06/03/2004
15	002779200	05/02/2004	16193	02MAY2004	5	May	05/02/2004	May	02MAY2004	05/02/2004
16	002779804	05/14/2004	16205	14MAY2004	5	May	05/14/2004	May	14MAY2004	05/14/2004

The point of this exercise is to demonstrate that (a) no matter how a SAS date is formatted, the data being stored is always the same integer value indicating the number of days between that date and January 1, 1960 (b) it is easy for text strings to be mistaken for date values so you need to make sure your date variable really is a numeric date before using it in a date calculation, and (c) just as the INPUT function can be used to convert dollar figures or resolve user-defined formats to numeric values, it can also be used to convert text representations of dates to numeric date values.

This last point is particularly important because you will receive dates in a wide variety of text-string forms. Unfortunately, in addition to receiving dates as text strings (most of which can be converted to numeric dates without too much trouble using the INPUT function) you might also receive dates as separate fields containing the month, day, and year associated with a specific event. In this case, you first need to create a single date field from the constituent parts before you can include the date variable in a calculation. In the following example, a file containing string variables for bill_month, bill_day, and bill_year is processed to produce a single "bill_date" variable using the MDY function.

```
DATA work.claims;
   SET hca.claims;
      bill_date = MDY(INPUT(bill_month,2.),INPUT(bill_day,2.),INPUT(bill_year,4.));
RUN;
```

The MDY function takes as its input three numeric values representing the month, day, and year that contribute to the date. In the preceding example, note that because these three variables are text strings, they were converted to numeric values using the INPUT function. It should also be noted that had the INPUT function not been used to convert the string variables (as in the following example) they would have still been converted to numeric values implicitly by SAS, but a note would be generated to the SAS LOG.

```
DATA work.claims;
   SET hca.claims;
      bill_date = MDY(bill_month,bill_day,bill_year);
RUN;
```

```
NOTE: Character values have been converted to numeric values at the places given
      by: (Line):(Column).  1758:18   1758:29   1758:38
```

This warrants mention because even though, in this case, the bill_date is still computed correctly, this computation happens as somewhat of a happy accident. SAS assumes that if you are using MDY, you intend for the three parameters passed to MDY to be numeric values—even though the columns passed were strings—so it attempts to convert the content of these strings to numeric values. Notes might also be written to the log as a result of the DATA step in which the INPUT function is <u>correctly</u> used to <u>explicitly</u> convert the three string variables to numeric input to the MDY function, but in this latter case those notes are letting you know that missing values are generated as a result of operation on missing values.

```
1964  bill_date = MDY(INPUT(bill_month,2.),INPUT(bill_day,2.),INPUT(bill_year,4.));
1965  RUN;
```

```
NOTE: Missing values were generated as a result of performing an operation on
      missing values.  Each place is given by: (Number of times) at
     (Line):(Column).
     31554 at 1964:13
```

Many SAS programmers become complacent with regard to these messages realizing that they are often encountered when SAS is performing implicit operations on your behalf or because there are some missing values in your dataset. Having already profiled the data and finding the results within your tolerances, you might easily dismiss these notes; they are not "errors" after all. But consider the current situation in which you are processing three variables to create a single date. It is entirely possible that the source system from which the data were extracted allows users to specify a month and year, but omit the "day". In the preceding example, you might very well have 31154 records that contain valid month and year values that could provide important analytic trend data. If you simply dismiss the generation of missing values and chalk it up to "yep, nothing you can do about missing data", you have lost potentially valuable information. As a general rule, when writing production-level code, you should make sure that your code does not generate such messages. For example, in the case of missing "day" data, you might write your code as follows—inserting a fixed value of "1" or "15" as the day used to generate the date[1].

---

[1] Note, however, that whether you actually use a static value should be driven by the analytic use of these data and requires stakeholder/end-user input regarding both the appropriateness of this approach and the static value that should be assigned.

```
DATA work.claims;
   SET hca.claims;
 IF bill_day = '' THEN
    bill_date = MDY(INPUT(bill_month,2.),1,INPUT(bill_year,4.));
 ELSE
    bill_date = MDY(INPUT(bill_month,2.),INPUT(bill_day,2.),INPUT(bill_year,4.));
RUN;
```

Even if you do not use conditional logic to assign a static value for the missing date component, however, you should still get in the habit of conditionally processing such data in a manner that eliminates implicit conversions as doing so (a) demonstrates that you have programmatic control over your data and mastery of the transformations you are performing and (b) helps keep your log cleaner—allowing you to more rapidly identify issues that do require your attention.

DATE MATH & INTERVAL FUNCTIONS

With your date variables stored as actual numeric dates, you can start using them to calculate the passage of time between events. Most commonly this entails knowing the number of days between two events, and because SAS dates simply represent the number of days since a fixed point in time (1/1/1960), the difference between two SAS dates represents the number of days between those dates. In the following example, the "payment_lag" variable is calculated as the number of days between the date a bill is generated and the date payment is received. Note here, again, that a logic check is created so the payment_lag is only calculated if paid_date is not null. If we receive a note about operating against missing values in this case, we know that the bill_date is the offending variable.

```
DATA work.claims;
 SET hca.claims;
 IF paid_date NE . THEN payment_lag = paid_date - bill_date;
RUN;
```

In some cases, you want a variable like "payment_lag" to be calculated with respect to the current date in cases where a payment has not yet been received (i.e., where “paid_date” is missing). In this case, because you want to know the lag of all payments and not just those for which payment has been received, you might execute the following DATA step instead. In the absence of a value of "paid_date", the current date (obtained using the TODAY function) is used as the end-point of interest.

```
DATA work.claims;
 SET hca.claims;
 IF paid_date NE . THEN payment_lag = paid_date - bill_date;
                   ELSE payment_lag = TODAY() - bill_date;
RUN;
```

In addition to calculating the number of days that have passed between two events, you will sometimes need to measure time in other units (e.g., weeks, months, years, etc.). In the following example the INTCK function is used to determine how many billing cycles have passed since the “bill_date”. INTCK is perfect for this type of calculation as it "counts the number of times a boundary has been crossed" (Cody, 2004, p. 183). In this example, the variable "billing_cycles" is calculated as the number of times the boundary of Monday (WEEK.2, or the second day of each week) has been crossed since the bill was generated. For a given claim record, if a bill was generated on Friday and this DATA step was run the following Wednesday, the value of "billing_cycles" would be "1"—as the billing cycle boundary has been crossed once since the time the bill was generated. Using this function you could build conditional logic to set a "review" flag for those bills that have extended past eight billing cycles without a payment being received.

```
DATA work.claims;
 SET hca.claims;
 IF paid_date = . AND INTCK('WEEK.2',bill_date,TODAY()) > 8
    THEN review='Y';
RUN;
```

If, on the other hand you wanted to know what the date will be eight billing cycles from the bill_date, you could use the INTNX function "to return the date after a specified number of intervals have passed" (Cody, 2004, p 190).

```
DATA work.claims;
 SET hca.claims;
 planned_review_date = INTNX('WEEK.2',bill_date,8);
RUN;
```

Both INTNX and INTCHK are very powerful tools for creating reports in support of operations that occur on some fixed schedule. In addition to specific days of the week, intervals can be defined using a variety of forms (e.g., "MONTH2" for every two month period, "WEEK4.3" to indicate a time period every four weeks starting on Tuesday, etc.). As useful as these functions are, however, you need to be careful that you are using them as intended. For example, you could be tempted to use INTCK to calculate a health plan member's age at the time a service was rendered. However, using the following code, the age of someone with a date of birth of "12/30/1970" undergoing a physical exam on "1/3/2001" would be calculated as "31" instead of "30" because 31 "year boundaries" have been crossed between these two dates.

```
DATA work.claims;
 SET hca.claims;
 age = INTCK('YEAR',dob,svc_date);
RUN;
```

Calculating "age" is certainly one of the most widely discussed date calculations among SAS programmers. Of course calculating age as the difference between someone's date of birth and the target date and then dividing that number by 365.25 (to account for leap years) is a widely accepted method to generate an approximate age—although it is not perfectly accurate because it accounts for leap years even in situations where none have occurred. Delwiche and Slaughter (2010) suggest another method (using INTCK) which is accurate for integer values of age, but does not render precise decimal values. Using Slaughter & Delwiche's method for integer age using INTCK, you might compute age in the claims dataset as follows:

```
DATA work.claims;
 SET hca.claims;
 age = INT(INTCK('MONTH', dob, svc_date)/12);
 IF MONTH(dob) = MONTH(svc_date)
   THEN age = age - (DAY(dob)>DAY(svc_date));
RUN;
```

In this example, the number of 12-month periods between the two dates is calculated to yield the interval in years ("age"); then, if the dob and the svc_date are in the same month an assessment of the day of the month is subtracted from "age". If the day of the month in the "dob" variable is greater than the day of month in the "svc_date" variable, the expression evaluates to TRUE (1) and one is subtracted from “age”. If the day of the “dob” is less than or equal to the day of “svc_date”, the expression resolves to FALSE (0) and the integer “age” is not changed.

Another date function that deserves attention in a discussion of date arithmetic in SAS is DATEPART. Several source systems from which you might extract data will store dates, by default, as datetime variables. As mentioned previously, SAS stores datetime data as an integer representing the number of seconds since 1/1/1960 at midnight. As a result, when comparing the date "5/13/2008" to the datetime "07MAY2008:09:30:25", you will find that the latter is actually greater than the former. Because you are comparing intervals expressed in different units of measurement (days vs. seconds) the IF statement in the following code would evaluate to TRUE for a record with bill_date = "5/13/2008" and admit_date = "07MAY2008:09:30:25". May 7th is after May 13th!

```
DATA WORK.review_cases;
 SET hca.encounters;
 IF bill_date  <  admit_date THEN OUTPUT;
    (5/13/2008)    (07MAY2008:09:30:25)
      (17,665)          (1,525,771,825)
RUN;
```

Comparisons of "date" variables stored in different units of measurement can obviously cause a host of problems. In the preceding example, it is unlikely that there would be any records in "hca.encounters" that would not be output to "work.review_cases". The most appropriate solution in this case is to apply the DATEPART function to "admit_date" within the IF statement. DATEPART does just what its name implies; it takes the "date part" from a datetime variable. Specifically, it converts the number of seconds since 1/1/1960 at midnight into the number of days since 1/1/1960. You can also perform a rough, approximate check of this logic by dividing the datetime value by (60 [seconds] x 60 [minutes] x 24 [hours] = 86,400). Or, you can simply use the DATETIME function—which will give you a more efficient, precise conversion. Applying the DATEPART function, the values resolve as you expect and May 13th is once again after May 7th.

```
DATA WORK.review_cases;
 SET hca.encounters;
 IF bill_date  <  DATEPART(admit_date) THEN OUTPUT;
    (5/13/2008)    (07MAY2008:09:30:25)
      (17,665)             (17,659)
RUN;
```

One final method that you will need to use from time to time is the comparison of a date variable to a fixed date corresponding to some event—e.g., the date a new clinic opens at a hospital, the date a new data processing system goes online, etc. In the following example, the billing_date in the "claims" dataset will be compared to the date when a new claims processing system went online (10/1/2011) in order to perform reviews of the records generated by that system. In this case the value “01OCT2011” in the IF statement is referred to as a “date literal” because you are making a comparison to a static (or literal) value rather than the value of a “variable”. Specification of a date literal involves putting single quotes around the static date value and appending it with the letter "d" to indicate to SAS that your static reference is a date value.

```
DATA WORK.review_claims;
  SET hca.claims;
  IF bill_date >= '01OCT2011'd THEN OUTPUT;
RUN;
```

STRING FUNCTIONS

One of the most frustrating tasks in creating a data product is extracting data elements that are buried within a character variable. Suppose, for example, that you are asked to create a subset of records from all hospital visits that occurred at the "skin clinic", but when you ask which variable indicates clinic location you find out that the abbreviation indicating the clinic name (SKNC) is part of the "location" field—which has entries like "SKNC-09-003", "SKNC-10-001", etc. In order to identify all such hospital encounters, you need to use the SUBSTR function—which extracts from a string variable the contents of the character positions specified as inputs. In the following example, the "location" variable is examined to find all of the records for which the four characters read starting at position one are "SKNC".

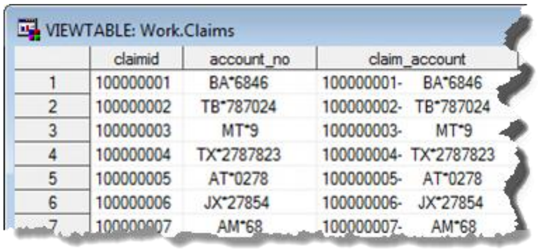
```
DATA WORK.skincenter_encounters;
  SET hca.claims;
  IF SUBSTR(location,1,4) = 'SKNC' THEN OUTPUT;
RUN;
```

It is nice when a character variable has information arranged in fixed positions like the preceding example, but often it is necessary to nest functions within one another such that the result of one function becomes an input to another function. This is required in many situations where functions are used to transform data, but seems especially true when trying to force character variables to give up the information they contain. Suppose the hospital from the previous example installed a new electronic medical record system, and in the process, made changes to the structure of the "location" variable—resulting in the "SKNC" designation being somewhat arbitrarily tossed around within "location" (e.g., 1BLUE*SKNC-09-003, 5YELLOW*SKNC-10-001, etc). Using the previous SUBSTR function call against this new variable would yield the values of "1BLU" and "5YEL", respectively, for these two records and miss the fact that both encounters were at the skin center. The key to successfully finding the location code, in this case is finding the position of "*" and dynamically substringing the new location variable from that point. In order to achieve this feat, the INDEX function is used to find the position of the first occurrence of the character "*" that tells us that the clinic name follows immediately, and that position is used as the starting position for the SUBSTR function.

```
DATA WORK.clinic_encounters;
 SET hca.encounters;
  IF = SUBSTR(location,INDEX(location,'*')+1,4) THEN OUTPUT;
RUN;
```

For a record with a "location" of "1BLUE*SKNC-09-003" the result of the INDEX function call is "6". Because the clinic code follows immediately after the position identified by the INDEX function, "1" is added to this result so that the call to the SUBSTR function becomes "SUBSTR(location,**7**,4)". The SUBSTR function for this record then reads the contents of "location" from position 7, for 4 characters. Just as in an arithmetic expression, the function calls resolve from the innermost set of parentheses to the outermost parentheses—with the result of the inner function becoming an input to the next outer function that it is nested within. Once you understand the mechanics of nested functions and begin to gain experience with the individual functions, nested functions almost become another data management language unto themselves, allowing you to do some pretty incredible things with text strings. Most of the string functions you will use (like SUBSTR, INDEX, etc.) are fairly easy to learn and remember because their output is very easy to recognize. You can see that "*" is in the sixth position of "location" in the previous example and you can visually validate that the SUBSTR of location from position 7, for 4 characters, should be "SKNC". However, when working with string functions that manage blank spaces within a character string, validation of the function is not as straight-forward simply because the result is not always what it appears to be.

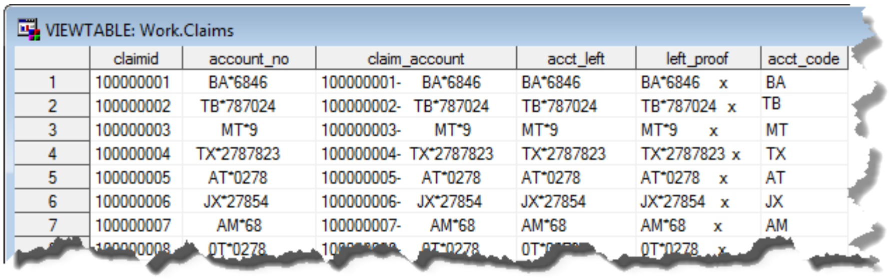
There are a number of reasons why you might need to manage white space within a character variable. Suppose you received a dataset with one or more variables containing leading or trailing spaces—like "account_no" (account number) in the following "claims" dataset. In addition to being difficult to review visually, the leading spaces cause other problems as well; sorting by "account_no" will probably not yield the expected results—as "   BM*48" will sort before " AJ*0027" and placement of the account number on reports and other output will vary as well. In addition, concatenating the account number with, for example, "claimid" to produce a new field can yield undesirable results as depicted in "claim_account".


VIEWTABLE: Work.Claims

	claimid	account_no	claim_account
1	100000001	BA*6846	100000001- BA*6846
2	100000002	TB*787024	100000002- TB*787024
3	100000003	MT*9	100000003- MT*9
4	100000004	TX*2787823	100000004- TX*2787823
5	100000005	AT*0278	100000005- AT*0278
6	100000006	JX*27854	100000006- JX*27854
7	100000007	AM*68	100000007- AM*68

Fortunately, there are a number of ways to manage blank spaces within a character variable. In the following DATA step, "acct_left" uses the LEFT function to move the leading blanks from the beginning of the "account_no" variable. Note that the blank spaces are not "removed" from the variable. The new variable still contains the blank spaces, but they are at the end of the content instead of the beginning. The dataset browser, however, does not render the blank spaces, so you cannot go into the "acct_left" field and count the spaces at the end of the variable to validate this claim. You can, however, concatenate a character (x) to the end of "acct_left" to prove that the blank spaces were moved and not deleted; the result is the variable "left_proof". The result of applying the LEFT function to "account_no" yields a variable that is a bit easier to browse and, more importantly, can be easily SUBSTRinged (e.g., to get the two letter "account_code") and sorted.

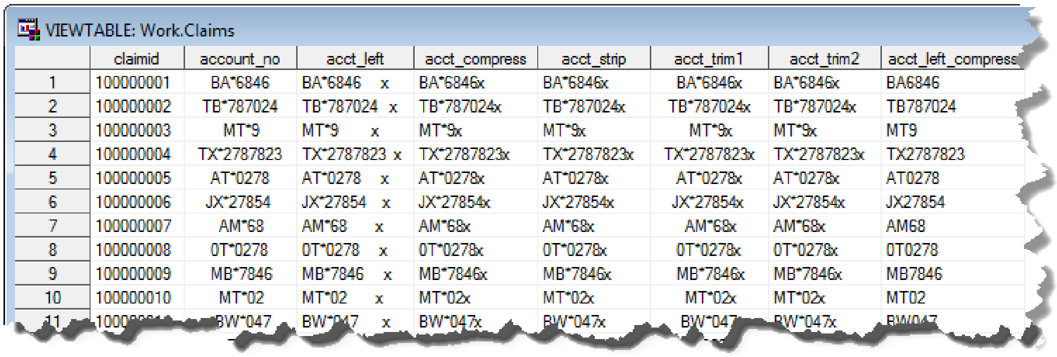
```
DATA work.claims;
 SET hca.claims;
    claim_account = claimid||'-'||account_no;
    acct_left = LEFT(account_no);
    left_proof = LEFT(account_no)||'x';
    acct_code=SUBSTR(acct_left,1,2);
keep claimid account_no claim_account acct_left left_proof acct_code;
RUN;
```


VIEWTABLE: Work.Claims

	claimid	account_no	claim_account	acct_left	left_proof	acct_code
1	100000001	BA*6846	100000001- BA*6846	BA*6846	BA*6846 x	BA
2	100000002	TB*787024	100000002- TB*787024	TB*787024	TB*787024 x	TB
3	100000003	MT*9	100000003- MT*9	MT*9	MT*9 x	MT
4	100000004	TX*2787823	100000004- TX*2787823	TX*2787823	TX*2787823 x	TX
5	100000005	AT*0278	100000005- AT*0278	AT*0278	AT*0278 x	AT
6	100000006	JX*27854	100000006- JX*27854	JX*27854	JX*27854 x	JX
7	100000007	AM*68	100000007- AM*68	AM*68	AM*68 x	AM
	100000008	0T*0278	[illegible]	[illegible]	0T*0278 x	

For cases in which you wish to remove the leading and trailing blanks altogether, you can apply the STRIP or COMPRESS functions. In the following example, STRIP and COMPRESS yield the same result (acct_strip and acct_compress, respectively)—removing all leading and trailing blanks. STRIP is specifically geared toward stripping leading and trailing blanks, whereas COMPRESS is more specialized—"compressing" character strings by removing specified characters no matter where they appear in the character string. When COMPRESS is used with no characters specified for removal, only blank spaces are removed. However, as shown in "acct_left_compress", if specific characters are indicated for removal all instances of those characters are removed. Unlike the use of STRIP and COMPRESS in this example, TRIM removes only trailing blanks and leaves leading blanks intact. Note that with TRIM applied to "account_no", the result "acct_trim1" still has leading blanks, however when TRIM is applied to the result of LEFT(account_no), the trailing blanks that result from application of the LEFT function are trimmed in the creation of "acct_trim2".

```
DATA work.claims;
 SET hca.claims;

    acct_left      = LEFT(account_no)||'x';
    acct_compress = COMPRESS(account_no)||'x';
    acct_strip     = STRIP(account_no)||'x';
    acct_trim1      = TRIM(account_no)||'x';
    acct_trim2      = TRIM(LEFT(account_no))||'x';
    acct_left_compress = COMPRESS(acct_left,' *x');

KEEP claimid account_no acct_left  acct_compress acct_strip
     acct_trim1 acct_trim2 acct_left_compress;
RUN;
```



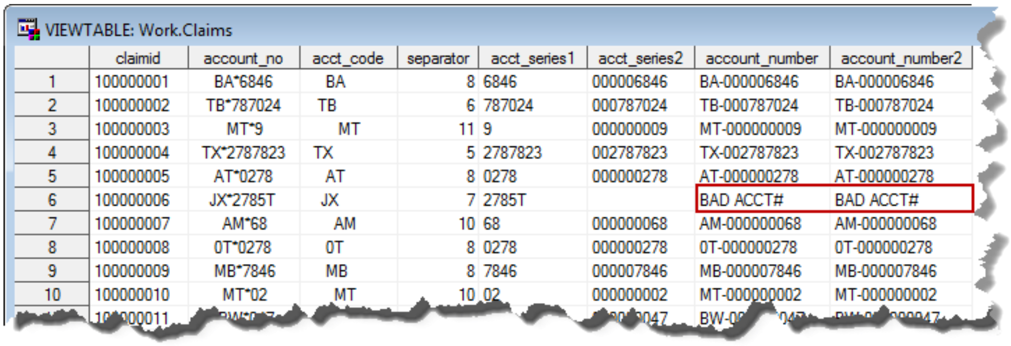
VIEWTABLE: Work.Claims

	claimid	account_no	acct_left	acct_compress	acct_strip	acct_trim1	acct_trim2	acct_left_compress
1	100000001	BA*6846	BA*6846 x	BA*6846x	BA*6846x	BA*6846x	BA*6846x	BA6846
2	100000002	TB*787024	TB*787024 x	TB*787024x	TB*787024x	TB*787024x	TB*787024x	TB787024
3	100000003	MT*9	MT*9 x	MT*9x	MT*9x	MT*9x	MT*9x	MT9
4	100000004	TX*2787823	TX*2787823 x	TX*2787823x	TX*2787823x	TX*2787823x	TX*2787823x	TX2787823
5	100000005	AT*0278	AT*0278 x	AT*0278x	AT*0278x	AT*0278x	AT*0278x	AT0278
6	100000006	JX*27854	JX*27854 x	JX*27854x	JX*27854x	JX*27854x	JX*27854x	JX27854
7	100000007	AM*68	AM*68 x	AM*68x	AM*68x	AM*68x	AM*68x	AM68
8	100000008	0T*0278	0T*0278 x	0T*0278x	0T*0278x	0T*0278x	0T*0278x	0T0278
9	100000009	MB*7846	MB*7846 x	MB*7846x	MB*7846x	MB*7846x	MB*7846x	MB7846
10	100000010	MT*02	MT*02 x	MT*02x	MT*02x	MT*02x	MT*02x	MT02
11	1000	BW*047	BW*047 x	BW*047x	BW*047x	BW*047	BW*047x	BW047

As mentioned previously, nesting several functions is often necessary to yield the desired outcome. It is not uncommon to get a dataset with a variable like "account_no" only to find out that the other analytic datasets have a slightly different form of account_no. If you want to join your data to those other datasets by account number, it will be necessary to get the account numbers in the two systems to match each other. In the following example, suppose you found out that the value of the account number corresponding to "BA*6846" in all of the other analytic datasets is "BA-000006846" That is, the two-digit account code (BA) is separated from the account series number (6846) by a hyphen (-) instead of an asterisk (*) and the series number needs to be left-padded with zeroes to a length of nine characters.

To perform such transformations, it is sometimes easier to break the values down into their constituent parts then join them back together in the desired form. One of the first things you need to know in this example is the location at which the "account code" is separated from the "acct series". This happens at the asterisk, so a variable called "separator" is created with the INDEX function to hold the number corresponding to the position of the asterisk within each account number. Next this location is used to extract the account code (acct_code) and account series (acct_series1) using the SUBSTR function. Next, a conditional logic control is put in the DATA step to control the creation of the transformed account number. Because the account series should contain only numbers, a check is performed to make sure that "acct_series1" only contains blank space and numeric characters. If "acct_series1" passes this test then the value of "acct_series2" is populated with a zero-filled version of "acct_series1" and the account number is written by COMPRESSING the concatenation of "acct_code" and "acct_series", separated by a hyphen. If "acct_series1" contains anything other than numeric characters and blank spaces, the value of "account_number" is assigned as "BAD ACCT#". Also in this DATA step, you will notice that there is a variable "account_number2" being written. This is simply the computation of the account number created by nesting all of the necessary functions within one another. The down-side of this latter approach (besides its complexity) is that you cannot perform the logic-check necessary to keep bad account numbers from being created, and to the extent that you encounter non-numeric characters in the account series, your log will be filled with notes indicating an invalid argument is being passed to the INPUT function.

```
DATA work.claims;
 SET hca.claims;
   separator = INDEX(account_no,'*');
   acct_code = SUBSTR(account_no,1,separator-1);
   acct_series1 = SUBSTR(account_no,separator+1);
   IF VERIFY(acct_series1,' 0123456789') = 0 THEN DO;
      acct_series2 = PUT(INPUT(acct_series1,9.),Z9.);
      account_number = COMPRESS(acct_code||'-'||acct_series2);
   END;
   ELSE account_number = 'BAD ACCT#';

  account_number2 = COMPRESS(SUBSTR(account_no,1,INDEX(account_no,'*')-1)||'-
'||PUT(INPUT(SUBSTR(account_no,INDEX(account_no,'*')+1),9.),Z9.));

   IF LENGTH(account_number2) ne 12 THEN
      account_number2 = 'BAD ACCT#';

KEEP claimid account_no separator acct_code acct_series1 acct_series2
     account_number account_number2;

RUN;
```



VIEWTABLE: Work.Claims

	claimid	account_no	acct_code	separator	acct_series1	acct_series2	account_number	account_number2
1	100000001	BA*6846	BA	8	6846	000006846	BA-000006846	BA-000006846
2	100000002	TB*787024	TB	6	787024	000787024	TB-000787024	TB-000787024
3	100000003	MT*9	MT	11	9	000000009	MT-000000009	MT-000000009
4	100000004	TX*2787823	TX	5	2787823	002787823	TX-002787823	TX-002787823
5	100000005	AT*0278	AT	8	0278	000000278	AT-000000278	AT-000000278
6	100000006	JX*2785T	JX	7	2785T		BAD ACCT#	BAD ACCT#
7	100000007	AM*68	AM	10	68	000000068	AM-000000068	AM-000000068
8	100000008	0T*0278	0T	8	0278	000000278	0T-000000278	0T-000000278
9	100000009	MB*7846	MB	8	7846	000007846	MB-000007846	MB-000007846
10	100000010	MT*02	MT	10	02	000000002	MT-000000002	MT-000000002

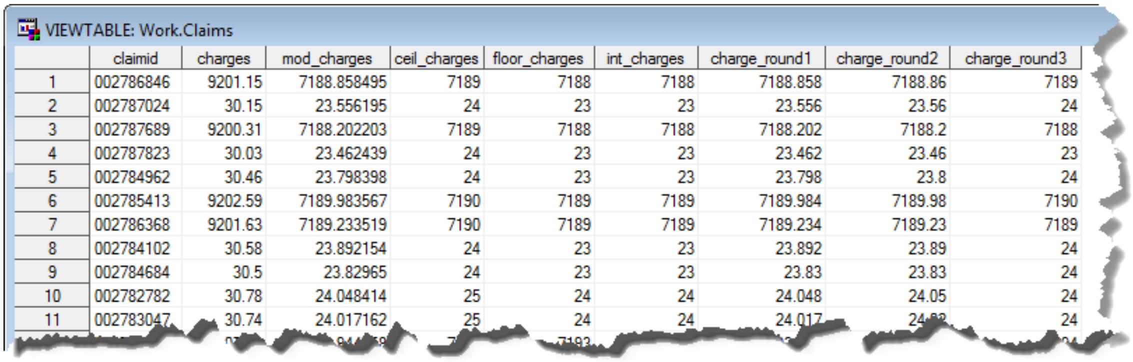
There are countless other transformations that you will need to perform in creating analytic datasets, but hopefully these examples will help you think about approaches to these interesting programming challenges.

NUMERIC FUNCTIONS

The final category of functions that will be discussed briefly are those involving the transformation of numeric data. Fortunately, compared to string and date functions, you will normally need to do considerably less with these functions in order to produce analytic datasets. Usually, beyond converting character strings to numeric values using the INPUT function and an INFORMAT, there are only a few functions you will probably use with any regularity—ROUND, INT, CEIL, and FLOOR.

In the following example, CEIL, FLOOR, and INT are all applied to the product of a charge modifier and the total charges associated with a claim. As you would expect, CEIL rounds this product up to the next higher integer, FLOOR rounds the value down to the next lower integer, and INT returns only the integer portion of the value.

```
DATA work.claims;
 SET hca.claims;

   mod_charges    = charges*.7813;
   ceil_charges   = CEIL(mod_charges);
   floor_charges  = FLOOR(mod_charges);
   int_charges    = INT(mod_charges);
   int_age_calc   = INT((INPUT(bill_date,yymmdd10.) - dob)/365.25);
  int_age_yrdif   = INT(YRDIF(dob,INPUT(bill_date,yymmdd10.),'ACTUAL'));
  charge_round1   = ROUND(mod_charges,.001);
  charge_round2   = ROUND(mod_charges,.01);
  charge_round3   = ROUND(mod_charges,1);

KEEP claimid charges mod_charges ceil_charges floor_charges int_charges
     charge_round1 charge_round2 charge_round3;
RUN;
```



	claimid	charges	mod_charges	ceil_charges	floor_charges	int_charges	charge_round1	charge_round2	charge_round3
1	002786846	9201.15	7188.858495	7189	7188	7188	7188.858	7188.86	7189
2	002787024	30.15	23.556195	24	23	23	23.556	23.56	24
3	002787689	9200.31	7188.202203	7189	7188	7188	7188.202	7188.2	7188
4	002787823	30.03	23.462439	24	23	23	23.462	23.46	23
5	002784962	30.46	23.798398	24	23	23	23.798	23.8	24
6	002785413	9202.59	7189.983567	7190	7189	7189	7189.984	7189.98	7190
7	002786368	9201.63	7189.233519	7190	7189	7189	7189.234	7189.23	7189
8	002784102	30.58	23.892154	24	23	23	23.892	23.89	24
9	002784684	30.5	23.82965	24	23	23	23.83	23.83	24
10	002782782	30.78	24.048414	25	24	24	24.048	24.05	24
11	002783047	30.74	24.017162	25	24	24	24.017	24.02	24

ADVANCED TECHNIQUES & OPERATIONALIZATION

With well-cleaned data and the ability to build and utilize user-defined formats, perform conditional logic checks and use a wide array of functions, you can perform a large number of data transformations on individual dataset records. However, a significant proportion of the work that goes into producing an analytic dataset involves rolling up datasets to a different level of granularity or computing summary statistics on subgroups of a larger dataset. Whereas PROC SQL is often a better tool for some of that work, it is also the case that a well-planned DATA step can accomplish multiple necessary transformations in a single pass through the data[2].

PROCESSING SORTED DATA & THE RETAIN STATEMENT

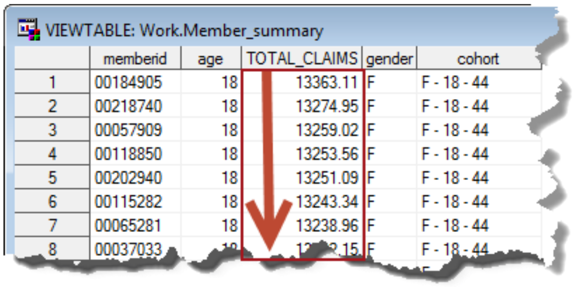
So far the data transformations that have been discussed are those that involve operating against one or more data elements on an individual record and transforming them, for example with an INPUT, PUT, DATEPART, or SUBSTR function, into something that is more informative or user-friendly than the raw data with which you started. Between that ability and PROC SQL, you have the ability to create some very powerful data management solutions. One type of transformation that can remain cumbersome with that set of tools, however, is computing a data element in one record based on inputs from another record. For example, think about how you might assign ranks to records based on some metric like "total_charges". Of course, in real life you would likely use PROC RANK, but for demonstration purposes, suppose you were assigning ranks to individual health plan members based on the total charges their health care has generated over the past year. Suppose, further, that you wanted to perform this ranking within certain subgroups—for example, an age/gender cohort.

The first thing you would want to do is sort the dataset by the values of the variable(s) on which you want to base the ranking. In the following example, a membership dataset with the total claims generated by each member is sorted by the member's age/gender cohort and in descending order by the total_claims associated with each member.

---

[2] For information on PROC SQL, readers are referred to the excellent tutorials provided by Lafler (2003, 2004a, 2004b, 2005) as well as in-depth paper on PROC SQL for data management (Schacherer & Detry, 2010).

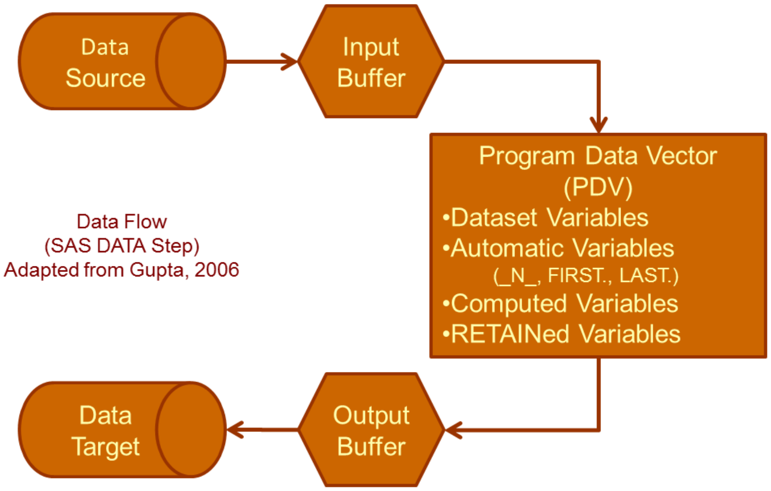
```
PROC SORT DATA=work.member_summary;
  BY cohort DESCENDING total_claims  ;
RUN;
```



VIEWTABLE: Work.Member_summary

	memberid	age	TOTAL_CLAIMS	gender	cohort
1	00184905	18	13363.11	F	F - 18 - 44
2	00218740	18	13274.95	F	F - 18 - 44
3	00057909	18	13259.02	F	F - 18 - 44
4	00118850	18	13253.56	F	F - 18 - 44
5	00202940	18	13251.09	F	F - 18 - 44
6	00115282	18	13243.34	F	F - 18 - 44
7	00065281	18	13238.96	F	F - 18 - 44
8	00037033	[illegible]	[illegible]	F	F - 18 - 44

Once sorted, the data can be processed in a DATA step using the same by-group variable(s). In the present example, doing so exposes two automatic variables available to SAS as the data are being processed—"FIRST.cohort" and "LAST.cohort". These automatic variables are only available in SAS's working memory; they are not part of any physical dataset[3]. As a record is read from the input buffer, the variables on the record are held in memory along with new computed variables being created during the DATA step and automatic variables such as _N_ (the record number within the dataset), and (when the data are processed using by-group variables) one or more sets of FIRST./LAST. variables. For each by-group variable in the DATA step, there is a FIRST.<by-group variable> and a LAST.<by-group variable>--each containing a boolean value indicating whether the current record is the first or last record of that by-group.



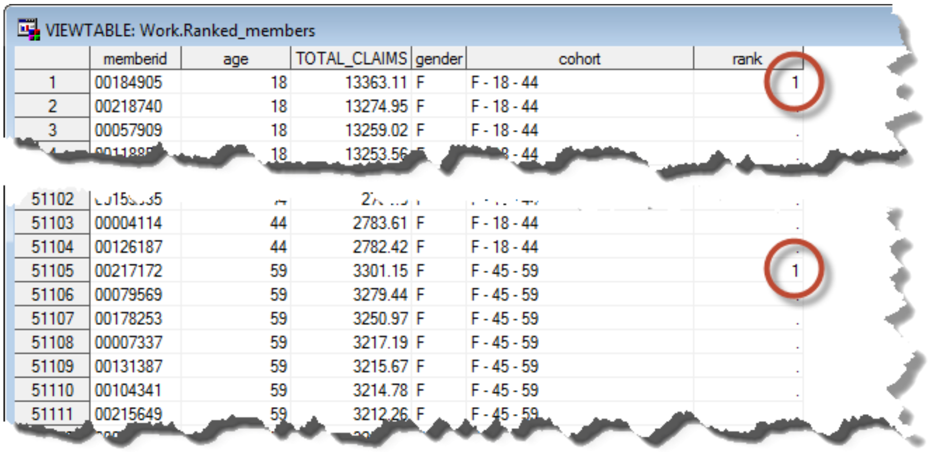
Data Flow
(SAS DATA Step)
Adapted from Gupta, 2006

In the following DATA step, the "FIRST.cohort" variable (exposed to the DATA step because "member_summary" is being processed by "cohort" is used to assign the value "1" to "rank" every time the "FIRST.cohort" variable evaluates to "TRUE". As shown in the depiction of the "ranked_members" dataset, below, this occurs only for the first record of each cohort; the "rank" value for other members of the cohort are missing despite the fact that the code clearly specifies that they should be assigned the value of "rank + 1".

```
DATA ranked_members;
 SET member_summary;
  BY cohort DESCENDING total_claims;
  IF FIRST.cohort THEN rank=1;
     ELSE rank = rank + 1;
RUN;
```

---

[3] Whereas this statement is true in that the automatic variables themselves are not written to a physical dataset, you can assign their values to local variables that can be written to a dataset. For example, assigning the local variable "rownum" the value of "_N_" will produce a value of rownum that contains the record number within the dataset being generated.



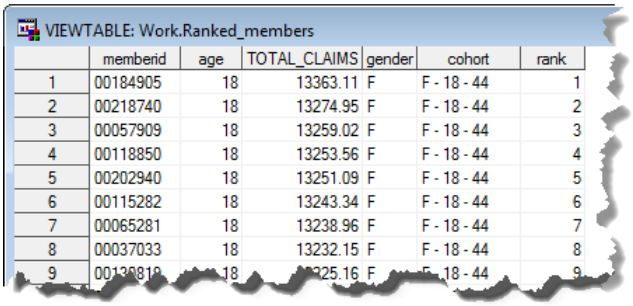
VIEWTABLE: Work.Ranked_members

	memberid	age	TOTAL_CLAIMS	gender	cohort	rank
1	00184905	18	13363.11	F	F - 18 - 44	1
2	00218740	18	13274.95	F	F - 18 - 44	.
3	00057909	18	13259.02	F	F - 18 - 44	.
[illegible]	[illegible]	18	13253.56	[illegible]	[illegible] - 44	.
51102	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	.
51103	00004114	44	2783.61	F	F - 18 - 44	.
51104	00126187	44	2782.42	F	F - 18 - 44	.
51105	00217172	59	3301.15	F	F - 45 - 59	1
51106	00079569	59	3279.44	F	F - 45 - 59	.
51107	00178253	59	3250.97	F	F - 45 - 59	.
51108	00007337	59	3217.19	F	F - 45 - 59	.
51109	00131387	59	3215.67	F	F - 45 - 59	.
51110	00104341	59	3214.78	F	F - 45 - 59	.
51111	00215649	59	3212.26	F	F - 45 - 59	[illegible]

The problem, of course, is that because "rank" is only assigned a non-null value for the first record of each cohort, the ELSE clause of the IF/THEN statement is trying to add "1" to a missing value for all other records in the cohort.—resulting in a missing value. Where is the value of "rank" to which SAS should be adding "1"? It is in the previous record. Because the DATA step flushes the values from the previous record from memory when a new record is fetched for processing from the input buffer, the value of "rank" that you would like to use in computing "rank" for the current record is no longer available when you want to use it. This is precisely the type of situation that can be resolved using the RETAIN statement.

The RETAIN statement keeps the values of specified variables from being automatically flushed from memory prior to the next record being loaded. In this way, if the next record being fetched from the input buffer does not have a value for that particular variable, the value retained from the previous record remains available to the DATA step for use in processing the new, current record. In the case of "rank" in the following DATA step, because it is retained from record to record, the value from the previous record is available for use in computing the value for the current record. When processing of the second record in the member_summary dataset begins, the value of "rank" (assigned during processing of the first record) is used to compute the value of "rank" for the second record, and so on.

```
DATA ranked_members;
 SET member_summary;
  BY cohort DESCENDING total_claims;
  RETAIN rank;
  IF FIRST.cohort THEN rank=1;
  ELSE rank = rank+1;
RUN;
```



VIEWTABLE: Work.Ranked_members

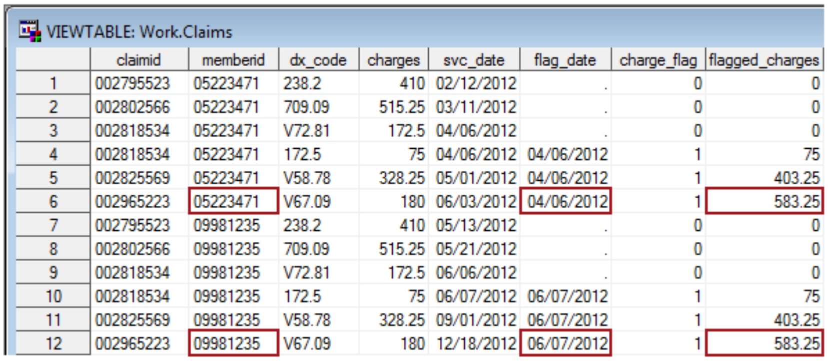
	memberid	age	TOTAL_CLAIMS	gender	cohort	rank
1	00184905	18	13363.11	F	F - 18 - 44	1
2	00218740	18	13274.95	F	F - 18 - 44	2
3	00057909	18	13259.02	F	F - 18 - 44	3
4	00118850	18	13253.56	F	F - 18 - 44	4
5	00202940	18	13251.09	F	F - 18 - 44	5
6	00115282	18	13243.34	F	F - 18 - 44	6
7	00065281	18	13238.96	F	F - 18 - 44	7
8	00037033	18	13232.15	F	F - 18 - 44	8
9	[illegible]	18	[illegible]	F	[illegible] 18 - 44	9

In the case of ties in the values of "total_claims", the RETAIN and IF statement logic can be extended as shown in the following DATA step to assign ties the same ranking. In this example, the variable "total_claims_r" is retained from one record to the next so that its value can be compared to the current value of "total_claims". If the value of the previous record and the current record are identical, the retained value of "rank" is not changed and that value is written to the current record. At the bottom of the DATA step, the "total_claims_r" variable is valued with the "total_claims" value from the current record so it can be used in the comparison that occurs when processing the next record in the dataset.

```
DATA ranked_members;
 SET member_summary;
  BY cohort DESCENDING total_claims;
  RETAIN rank total_claims_r;
  IF first.cohort then rank=1;
  ELSE IF total_claims NE total_claims_r THEN
       rank = rank+1;
  total_claims_r = total_claims;
RUN;
```

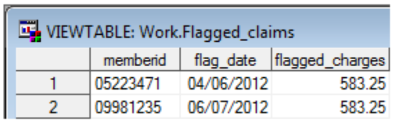
As was mentioned earlier, ranking records is more efficiently performed using PROC RANK, but if you are already processing ordered data and need to compute ranks, this method can be very useful. Beyond producing ranks, however, there are many cases in which you might want to use the RETAIN statement to assist you in summarizing your data. Perhaps you wanted to sum all claims generated by a health plan member following a specific diagnosis. You could do this with PROC SQL using a correlated sub-query, but depending on the size of your dataset, performance issues might come into play. If, on the other hand, you had already sorted your dataset by memberid and date, you could switch on a flag variable that indicates the event of interest has occurred. Then, during the processing of each subsequent record for that member, you could conditionally add the value of "charges" to a retained variable (flagged_charges) that gets incremented appropriately until the last record for that patient—at which point it gets written out to the target dataset (flagged_claims). In the following example, two datasets "claims" and "flagged_claims" are produced by processing the dataset "claim_details" by "memberid" and "svc-date". When the first record of each member is processed, the three retained variables "charge_flag", "flag_date" and "flagged_charges" are assigned the values "0", "null", and "0", respectively. Next, if at any point in processing that member's claims a diagnosis with "code=172.5" is encountered, the service date associated with that claim record is assigned to the retained variable "flag-date" and the value of the retained variable "charge_flag" is switched from "0" to "1". Each subsequently-processed record will contain a "flagged_charges" value corresponding to the running total of "charges" from that point through the last claim detail record associated with that member. If at the time the last record for that member is processed the "charge_flag" has a value of "1", the last record—with the running total "flagged_charges"—is output to the dataset "flagged_claims" to produce a dataset of the patients with this particular diagnosis and the total of the charges accumulated from the date of the diagnosis. Conversely, every record processed from "claim_details" will be output to the "claims" dataset.

```
DATA claims flagged_claims (KEEP = memberid flag_date flagged_charges);
 SET claim_details;
  BY memberid svc_date;

RETAIN charge_flag flag_date flagged_charges;

IF FIRST.memberid THEN do;
   flagged_charges=0;
   charge_flag = 0;
   flag_date=.;
   END;

IF code = '172.5' AND charge_flag = 0 THEN DO;
   flag_date=svc_date;
   charge_flag = 1;
   END;

IF charge_flag = 1 THEN flagged_charges = charges + flagged_charges;
IF LAST.memberid AND charge_flag=1 THEN OUTPUT flagged_claims;

OUTPUT claims;

RUN;
```



VIEWTABLE: Work.Claims

	claimid	memberid	dx_code	charges	svc_date	flag_date	charge_flag	flagged_charges
1	002795523	05223471	238.2	410	02/12/2012	.	0	0
2	002802566	05223471	709.09	515.25	03/11/2012	.	0	0
3	002818534	05223471	V72.81	172.5	04/06/2012	.	0	0
4	002818534	05223471	172.5	75	04/06/2012	04/06/2012	1	75
5	002825569	05223471	V58.78	328.25	05/01/2012	04/06/2012	1	403.25
6	002965223	05223471	V67.09	180	06/03/2012	04/06/2012	1	583.25
7	002795523	09981235	238.2	410	05/13/2012	.	0	0
8	002802566	09981235	709.09	515.25	05/21/2012	.	0	0
9	002818534	09981235	V72.81	172.5	06/06/2012	.	0	0
10	002818534	09981235	172.5	75	06/07/2012	06/07/2012	1	75
11	002825569	09981235	V58.78	328.25	09/01/2012	06/07/2012	1	403.25
12	002965223	09981235	V67.09	180	12/18/2012	06/07/2012	1	583.25



VIEWTABLE: Work.Flagged_claims

	memberid	flag_date	flagged_charges
1	05223471	04/06/2012	583.25
2	09981235	06/07/2012	583.25

Being able to compare data elements from different records to one another in the context of a DATA step opens another dimension to DATA step programming. No longer are you limited to sequentially processing each record and using only the data elements in that record to perform transformations, but you can now "borrow" information from other records to do meaningful comparisons across records or groups of records. Combined with other DATA step techniques such as hash objects (Dorfman, 2000; Dorfman & Vyverman, 2009), arrays (Cochran, 2009; Cody, 1999b), and the SAS macro language (Burlew, 1998; Carpenter, 2004) you will be able to efficiently perform nearly every type of transformation that can be done with BASE SAS. As the techniques and methods you master expand, you will likely be tapped to take on more and more complex projects, so the last few sections of the paper will focus on techniques that will help you operationalize production data management jobs.

MACRO VARIABLES & DRIVER PROGRAMS

As the complexity of your data product expands and the need to reliably execute your code for a given set of parameters comes into play, there are considerations that go beyond the types of transformations discussed so far. One important consideration is that your data management program be flexible enough to be rerun with different parameters month-after-month, quarter-after-quarter with minimal (or, ideally, no) manual intervention. For example, if you are developing a monthly process to download a data file and extract and transform the data it contains, you might use the following FILENAME statement to identify the file "2011_07.txt" and read it into the dataset "claims" using a DATA step that also filters the data to include only those records having a value "svc_date" in the date range bound by the first and last day of the month.

```
FILENAME source FTP  '2012_07.txt' USER='chris' PASS=&PASSWORD
         HOST = 'cdms-llc.com' CD ="ftproot\public\" DEBUG;

DATA claims;
   INFILE source FIRSTOBS=2 MISSOVER LRECL=300;
   INPUT @1 svc_date mmddyy10. @13 account_no $10. @25 prin_dx $6.
         @50 paid_date mmddyy10. @273 billed_charges dollar12.;
   WHERE '01JUL2012'd <= svc_date < '31JUL2012'd;
   <additional transformations>
RUN;
```

This program may have all of the transformations necessary to create the perfect dataset for your analytic needs, but every month when you want to run the program to get the previous month's data, you will need to go through the code and change all of those hard-coded values. You might argue that this is not a big deal because you can just use Find/Replace to make sure that all instances are changed. That probably will work nearly every time, but why risk it? Instead you could change the code to be more flexible and require less manual intervention. In the following adaptation of the previous code, three macro variables ("file_date", "start_date", and "stop_date") are added to the code to make it a bit more manageable. Here, you only need to specify the name of the file (file_date), start date, and stop date once, at the beginning of the program. Macro variable references throughout your code resolve to the assigned values and your program is a bit easier to manage.

```
%LET file_date= 2012_07;
%LET start_date= '01JUL2012'D;
%LET stop_date = '31JUL2012'D;
FILENAME source FTP  "&file_date..txt"  USER='chris' PASS=&PASSWORD
         HOST = 'cdms-llc.com' CD ="ftproot\public\" DEBUG;
DATA claims;
   INFILE source FIRSTOBS=2 MISSOVER LRECL=300;
   INPUT @1 svc_date mmddyy10. @13 account_no $10. @25 prin_dx $6.
         @50 paid_date mmddyy10. @273 billed_charges dollar12.;
   WHERE &start_date <= svc_date < &stop_date ;
<               additional transformations               >
RUN;
```

But even this solution requires manual intervention with the program in order to run it month to month. If you can assume that your source data file name always follows the naming convention "name_month.txt" you could automate this program to run each month as a scheduled batch job without any manual intervention whatsoever. Using the appropriate nested functions, you could schedule this program to run on the first day of each month to process the data file generated for the previous month. Specifically, in the example of July 2012 claims data, when run on August 1, 2012, the macro variables "src_file" resolves to "2012_07.txt". Passing this value to the FILENAME statement, the source file is dynamically specified without any changes to the program logic. Similarly, the IF/THEN logic is dynamically specified with the correct date range from "01JUL2012" to "31JUL2012".

```
%LET src_file = %SYSFUNC(YEAR(%SYSFUNC(INTNX(MONTH,%SYSFUNC(TODAY()),-1))))_
%SYSFUNC(MONTH(%SYSFUNC(INTNX(MONTH,%SYSFUNC(TODAY()),-1))),Z2.).txt;

%LET start_date = %SYSFUNC(INTNX(MONTH,%SYSFUNC(TODAY()),-1));
%LET stop_date  = %SYSFUNC(INTNX(MONTH,&START_DATE,1));

FILENAME source FTP  "&src_file" USER='chris' PASS=&PASSWORD
        HOST = 'cdms-llc.com' CD ="ftproot\public\" DEBUG;
DATA claims;
   INFILE source FIRSTOBS=2 MISSOVER LRECL=300;
   INPUT @1 svc_date mmddyy10. @13 account_no $10. @25 prin_dx $6.
         @50 paid_date mmddyy10. @273 billed_charges dollar12.;
   WHERE &start_date <= svc_date < &stop_date ;
<               additional transformations               >
RUN;
```

As the previous example demonstrates, the ability to dynamically specify source-file names and conditional values for your programs can greatly aid in automating programs, but such dynamic naming can also be a valuable aid in managing and distributing your final data product. One common goal of data management programs is the delivery of reports, SAS datasets, and data files in formats other than SAS (e.g., Microsoft Excel). Whether these data products are generated daily, weekly, monthly, or on an ad-hoc basis, it is imperative that they are identifiable with some sort of reference to the date (or datetime) when they were produced. One way to achieve this identification is to add a date or datetime stamp to the end of the filename of a product. In the following example, the "client summary" file is produced on an ad-hoc basis for service representatives who want information about the company's client portfolio. In order to facilitate communication about the "version" or "point in time" when the client summary was produced, you could automatically generate a datetime stamp to append to the filename, as in the following example.

```
%LET rpt_date = %SYSFUNC(PUTN(%SYSFUNC(DATE()),YYMMDD10.))
(%SYSFUNC(TRANSLATE(%SYSFUNC(PUTN(%SYSFUNC(TIME()),TIMEAMPM12.)),.,:)));

DATA _NULL_;
CALL SYMPUT('output_file1', "'" ||"f:\client_summary - "|| RESOLVE('&rpt_date')
||".xls'");
RUN;
```

```
PROC EXPORT DATA= WORK.client_summary
            OUTFILE= &output_file1
            DBMS=xls REPLACE;
     SHEET="client summary";
RUN;
```

When run at roughly 9:38 am on July 17, 2012, this method generates the excel file "client_summary - 2012-08-17 (9.38.58 PM).xls". Now, when someone has a question about the contents of the file, you can begin looking into their concern by understanding when the dataset or report was generated and miscommunications are less likely to lead to further confusion.

Another situation in which you might want to use this method is in the generation of LOG files for processes that you have scheduled to run as unattended batch jobs.

WRITING & READING THE LOG FILE

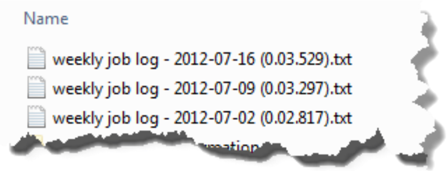
As a SAS programmer, one of the things that you have gotten used to doing is looking at your LOG window to confirm that your program ran without "`ERROR:`" notifications or "`NOTEs:`" indicating that implicit data type conversions have occurred. When programs are scheduled to run in an automated fashion, however, your chance to catch errors in an interactive fashion is lost. For this reason, when scheduling such jobs, you will probably want to generate an external log file that will persist after the SAS session has ended. That way, if you suspect that errors were generated by the job, you can still refer to the log file to see what happened. In the following example, PROC PRINTTO is used to write the log to the file "scheduled job log.txt".

```
FILENAME outlog 'f:\scheduled job log.txt';
PROC PRINTTO LOG = outlog;
RUN;
```

After your scheduled SAS job has finished running, you can refer to the .txt file to review the execution of your program. The problem with this approach, however, is that the log file name is static. As a result, if the execution of your SAS program does fail, you can review the log for errors that might have occurred, but you cannot compare the log to the log generated during the previous execution of the same program to see what happened differently during the failed execution. This is another situation in which you might want to dynamically assign a date-stamped name to an external file. In the following example, a date/time stamp is assigned to the log file specified in the execution of the PRINTTO procedure. Each time the program is executed, a log file specific to that execution is generated.

```
%LET log_date = %SYSFUNC(PUTN(%SYSFUNC(DATE()),YYMMDD10.))
(%SYSFUNC(TRANSLATE(%SYSFUNC(PUTN(%SYSFUNC(TIME()),HHMM10.3)),.,:)));

DATA _NULL_;
CALL SYMPUT('log_file', "'" ||"f:\weekly job log - "|| RESOLVE('&log_date')
||".txt'");
RUN;

FILENAME outlog "&log_file";
PROC PRINTTO LOG = outlog;
RUN;
```

For executions of this program on July 2nd, 9th, and 16th of 2012, the program will have generated log files like those shown below, and now you have a persistent copy of the log file from each execution of your program. That can be a very handy thing to have to help debug the occasional failure and is a necessity for production jobs that are mission-critical to your organization.



weekly job log - 2012-07-16 (0.03.529).txt
weekly job log - 2012-07-09 (0.03.297).txt
weekly job log - 2012-07-02 (0.02.817).txt

However, in addition to having an archived version of your log file that you can refer to after the fact in case of a job failure, you might also want to study your LOG file programmatically and use it as a control device for the process being executed. As described elsewhere (Schacherer & Steines, 2010), you can develop sophisticated methods for reading the log file from a SAS session and programming go/no-go decisions that control subsequent steps of your program. As a simple example, suppose you have a step in your program that e-mails a client a weekly report or truncates a database table prior to loading a new data mart fact table. You really want to make sure that the previous processing steps have executed properly, without error before those critical steps are executed. One simple way to assure that these steps did not generate errors is to read in the external log file and check for occurrences of the word "ERROR" in the beginning of a line. To expand on the previous example, you would first need to execute the PRINTTO procedure without a file reference to suspend writing to the file associated with "outlog". Next, you could read the "outlog" file into a SAS dataset ("log_data") and parse the file to find occurrences of "ERROR"—assigning the field "error_flag" the value of "1" for each line in which an "ERROR" is noted at the beginning of the line.

```
PROC PRINTTO;
RUN;

DATA work.log_data;
 INFILE outlog TRUNCOVER;
 INPUT logline $200.;
 IF SUBSTR(logline,1,5) = 'ERROR' THEN error_flag = 1;
RUN;
```

Next, with the error data captured in "log_data", you can count the number of errors encountered and control processing based on that value. In the following PROC SQL statement, output is suppressed with the NOPRINT option and the SUM (COUNT would also work in this case) of the values of "error_flag" in the "log_data" dataset is calculated and assigned as the value of the macro variable "num_errors".

```
PROC SQL NOPRINT;
 SELECT SUM(error_flag) INTO :num_errors
   FROM work.log_data
QUIT;
```

As described by Flavin & Carpenter (2002), the following DATA _NULL_ step is used to control processing based on whether errors were detected in the log file. If the value of "num_errors" is equal to zero, the macro variable "terminator" is assigned a null value. Conversely, if the number of errors recorded in the log file is greater than 0, the value of "terminator" is assigned a value corresponding to (a) the file syntax needed to send a failure message to the process administrator, followed by (b) an ENDSAS command. If no errors were written to the log file, when &terminator is later resolved following the DATA _NULL_ step, there is no executable code for SAS to process and processing of the remaining SAS syntax continues. If, on the other hand, errors were written to the LOG file "outlog", resolution of the "terminator" macro variable results in a job failure message being e-mailed to the process administrator with the subject line "Claims Job Failed – EOM"[4]. After this e-mail is sent, the SAS session is ended by the ENDSAS command.

```
DATA _NULL_;
 IF &num_errors = 0 THEN
 CALL SYMPUT('terminator','');
 ELSE
 CALL SYMPUT('terminator',"FILENAME job_done EMAIL to='cschacherer@cdms-llc.com'
subject='Claims Job Failed - EOM'; DATA _NULL_; FILE job_done; RUN; ENDSAS;");
RUN;

&terminator
```

---

[4] For more detailed information on e-mailing from SAS, readers are referred to Schacherer (2012) and Pagé (2004).

USING PROCESS AUDIT DATA TO CONTROL PROCESSING

In addition to encountering execution errors during the processing of your SAS logic, there are other reasons you might want to stop processing of your program before a critical step. For example if the "claims" source file usually represents $100,000 worth of claims and this month the processed file only contains $5,000 dollars in claims, this might be cause for concern. It is important to note that in this situation processing could very well complete without any syntax errors. For example, there might have been errors in the upstream process that generated the file for your use, or a system even further upstream may have been upgraded with new software that contained a serious flaw not discovered in previous testing. Or course, it could also be that there are external forces that conspired to wreak havoc on your business operations, and these data truly reflect the new reality for your organization, but you should err on the side of caution and assume that such a large deviation from the norm is cause to suspect corrupt data.

The first step in being able to control your program based on audit data is to collect the audit data in the first place. One way to do this is to create a SAS dataset of various audit metrics associated with your program and build the collection of these metrics into your program. For example when processing the dataset "claims", you might have an audit table called "audit_metrics" into which you could load the number of records found in the source data, the sum of the "total_charges" field, etc. For every successful execution of your program in the production environment, you will have information about the previous runs of the program. You can then use these data to inform subsequent executions of the program and determine at various points in the program's execution whether processing should continue or if you should be alerted to possible threats to the validity of the data.

```
PROC SQL;
INSERT INTO audit_metrics (process_date,metric_code,metric_name,measure)
 SELECT today(),10,'Number of Records',count(*)
   FROM work.claims;

INSERT INTO audit_metrics (process_date,metric_code,metric_name,measure)
 SELECT today(),11,'Sum of Total Charges',sum(total_charges)
   FROM work.claims;
QUIT;
```

Based on the data in the "audit_metrics" dataset, you can compare data associated with the current execution of your program to the historical trends to determine if the current data differ significantly from the data associated with previous executions. In the following example, the number of rows of data processed in the current run of the program is compared to the number of rows in the most recent execution of the program preceding the current one. The first step determines the number of records in the "claims" dataset during the last run of the program.

```
PROC SQL;
 SELECT measure INTO :claims_record_count_previous
   FROM audits.audit_metrics a
  WHERE metric_code = 10 AND
        process_date = (SELECT MAX(load_date)
                          FROM audits.audit_metrics);
QUIT;
```

After this record count is assigned to the macro variable :"claims_record_count_previous", the corresponding record count is derived from the dataset currently being processed and that value is assigned to "claims_record_count_current".

```
PROC SQL;
 SELECT COUNT(*) INTO :claims_record_count_current
   FROM work.claims;
QUIT;
```

The current and previous record counts are then compared to see if the current month's data falls within the tolerance levels determined by previous experience with these data. If the number of records in the current dataset is less than 95% (or above 105%) of the number of records from the previous month, the macro variable &audit_001 is assigned the value of '1'; if the number of records falls within the tolerance range, &audit_001 is assigned the value '0'.

```
DATA _NULL_;
 IF &claims_record_count_current < .95*(&claims_record_count_previous) OR
    &claims_record_count_current > 1.05*(&claims_record_count_previous) THEN
    CALL SYMPUTX('audit_001',1);
  ELSE CALL SYMPUTX('audit_001',0);
RUN;
```

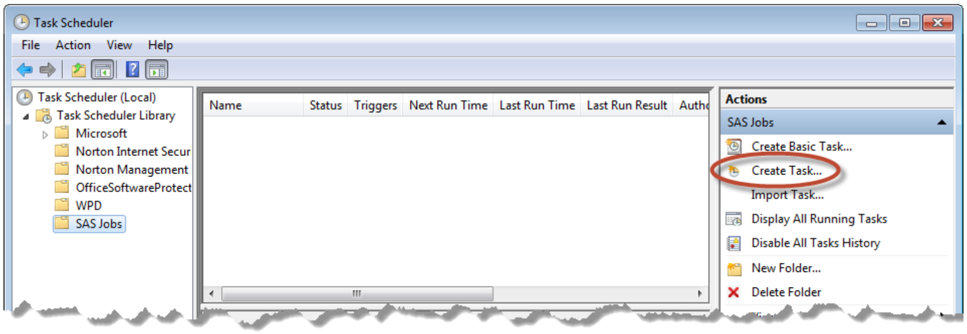
At this point, you could add a DATA _NULL_ step to evaluate "audit_001" and determine the course your program should take, or you could create several "audit_xxx" macro variables and generate an "audit_total" that could be similarly used to control processing.

```
DATA _NULL_;
CALL SYMPUTX('audit_total',&audit_001 + &audit_002 + ... &audit_xxx);
RUN;

DATA _NULL_;
 IF &audit_total = 0 THEN
 CALL SYMPUT('terminator','');
 ELSE
 CALL SYMPUT('terminator',"FILENAME job_done EMAIL to='cschacherer@cdms-llc.com'
subject='Claims Job Failed - EOM'; DATA _NULL_; FILE job_done; RUN; ENDSAS;");
RUN;

&terminator
```

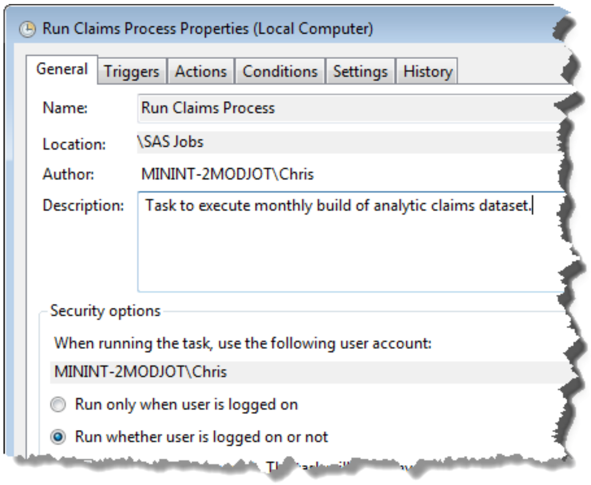
One note of caution is that the example shown here does not include logic to track the success or failure of program executions in the audit data or otherwise clean out the audit dataset in the case of failed executions. In the case of failed executions, the audit log should be cleaned up to make sure it contains only data from successful executions of the program so that future comparisons to the audit data are meaningful.

AUTOMATION – WINDOWS SCHEDULER

Of course, the goal of most production SAS programs is to have them run unattended on a set schedule. Therefore, once you have coded your data transformations, automated resolution of source file names, and implemented data-driven decision-points, all that is left to do is to schedule the program's recurring execution. Although there are a number of more sophisticated job control and scheduling software packages available, the Windows Task Scheduler® is widely available and is fairly simple to utilize for this purpose. In Windows 7, to launch the Task Scheduler, select Start➔Administrative Tools➔Task Scheduler.



After giving your new task a name and description go to the "Triggers" tab and specify the schedule on which you would like to run your program (e.g., weekly, daily, etc.). [In earlier versions of Task Scheduler, this step is performed on the "Schedule" tab.]



Run Claims Process Properties (Local Computer)

Name: Run Claims Process

Location: \SAS Jobs

Author: MININT-2MODJOT\Chris

Description: Task to execute monthly build of analytic claims dataset.

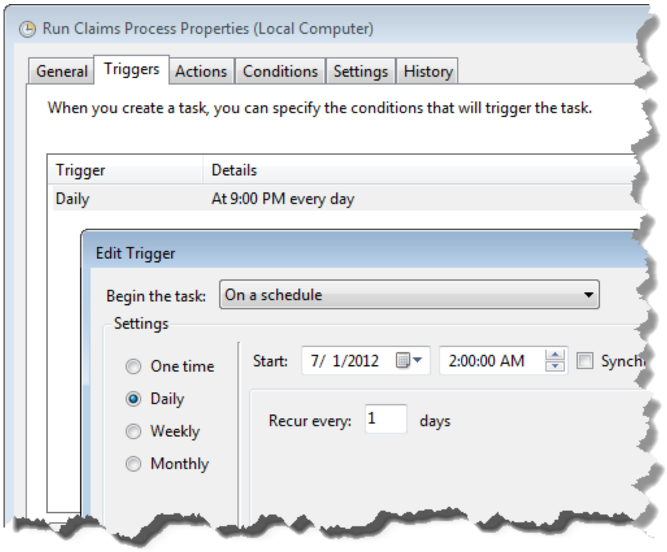
Security options

When running the task, use the following user account:

MININT-2MODJOT\Chris

○ Run only when user is logged on

● Run whether user is logged on or not



Run Claims Process Properties (Local Computer)

When you create a task, you can specify the conditions that will trigger the task.

Trigger	Details
Daily	At 9:00 PM every day

Edit Trigger

Begin the task: On a schedule

Settings

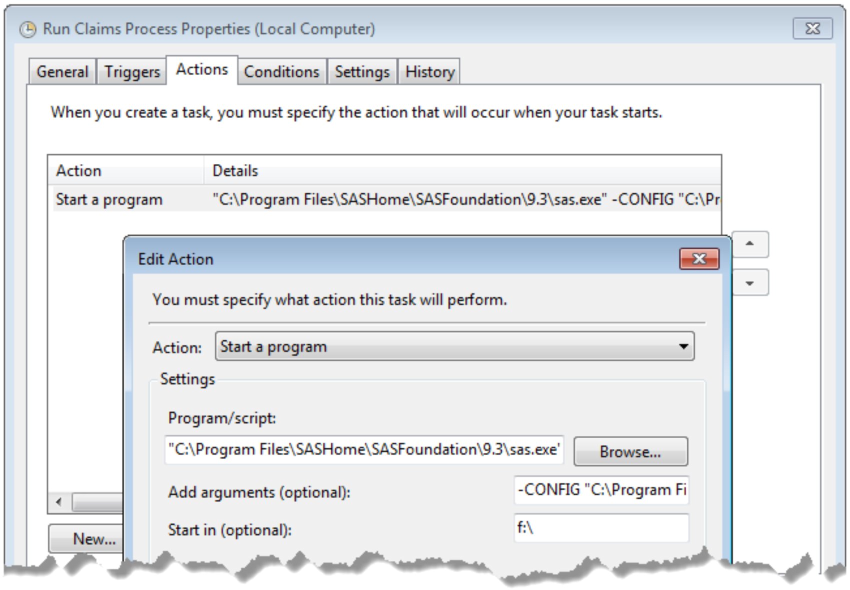
○ One time
● Daily
○ Weekly
○ Monthly

Start: 7/ 1/2012 2:00:00 AM ☐ Synch

Recur every: 1 days

In the "Actions" tab, you specify that you want the scheduled task to run "SAS.exe" and pass arguments specifying the SAS configuration file you wish to use (-CONFIG argument) as well as the SAS program to execute (-SYSIN argument). In this example the default SAS configuration file is specified and the –SYSIN argument directs the task to the program "create analytic claims dataset" located in the "F:\" drive.

-CONFIG "C:\Program Files\SASHome\SASFoundation\9.3\nls\en\SASV9.CFG"
-SYSIN "create analytic claims dataset.sas"



Run Claims Process Properties (Local Computer)

When you create a task, you must specify the action that will occur when your task starts.

Action	Details
Start a program	"C:\Program Files\SASHome\SASFoundation\9.3\sas.exe" -CONFIG "C:\Pr

Edit Action

You must specify what action this task will perform.

Action: Start a program

Settings

Program/script: "C:\Program Files\SASHome\SASFoundation\9.3\sas.exe'

Add arguments (optional): -CONFIG "C:\Program Fi

Start in (optional): f:\

The job is now scheduled to run every morning at 2:00 a.m. At that time, Windows will execute the schedule task, running the SAS program you specified to transform your source data into an analytic dataset. Depending on the logic you have specified, this scheduled job might e-mail a report to your supervisor, enter data into an audit dataset, use data from an audit dataset to make process-control decisions and deliver a date/time stamped dataset to a LAN directory. Because you took the time to make sure you started with a complete understanding of the data, its constraints, and short-comings, made transformations necessary to convert the data into easily-consumed information, and built solid process controls to assure its reliable delivery, you now have a reliable and valid data product of which you can proudly take ownership. With continued communication with the owners of upstream business and system processes and an understanding of the needs of your customers, you can manage the transformation of this product over time to make sure it remains a valuable asset to your organization.

FINAL THOUGHTS

Though somewhat lengthy for a user-group paper, the current work merely touches on a few of the important technical and solution development lifecycle topics necessary to a complete discussion of how to produce great data products. On the technical side of things, one of the most important SAS skills you should develop if you are working as a SAS data management professional is PROC SQL. Similarly important is the SAS MACRO language; this important tool helps encapsulate repetitive tasks so that they can be executed flexibly and reliably with different input parameters. Regarding the gathering of functional requirements and the solution development lifecycle for your data products, the author believes it matters less which methodology one follows (e.g., Agile, Waterfall, etc.) than it does clearly communicating to stakeholders and subject matter experts issues that will impact the data transformations you are planning in order to bring their solution to fruition. Too many data modelers take advantage of the relatively modest technical skills of their customers and "bully" them with technical jargon in order to code the solution the way they want to code it (and not necessarily in a manner that produces the product their clients need). This is a poor practice and often results in inadequate or even dangerously inaccurate solutions. Take the time to walk your clients, stakeholders, and subject matter experts through an explanation of the challenges you face in working with their data, propose the various solutions you can bring to bear on the issues, and clearly delineate the pros and cons of each approach. That way, these other contributors to your project will not only be able to help you make better decisions, but will also become more engaged in the success of your project.

REFERENCES

Burlew, M.M. (1998). SAS® Macro Programming Made Easy. Cary, NC: SAS Institute, Inc.

Carpenter, A. (2004). Carpenter's Guide to the SAS® Macro Language, Second Edition. Cary, NC: SAS Institute, Inc.

Cochran, B. (2009). Using SAS® Arrays to Manipulate Data. Proceedings of the SAS Global Forum 2009. Cary, NC: SAS Institute, Inc.

Cody, R. (1999a). Cody's Data Cleaning Techniques. Cary, NC: SAS Institute, Inc.

Cody, R. (1999b). Transforming SAS® Data Sets Using Arrays. Proceedings of the 24th Annual SAS Users Group International Meeting. Cary, NC: SAS Institute, Inc.

Cody, R. (2004). SAS® Functions by Example. Cary, NC: SAS Institute, Inc.

Cody, R. (2010). Using Advanced Features of User-Defined Formats and Informats. Proceedings of the SAS Global Forum 2010. Cary, NC: SAS Institute, Inc.

Delwiche, L.D. & Slaughter, S.J. (2010). The Little SAS Book: A primer, Fourth Edition. Cary, NC: SAS Institute, Inc.

DeMets, DL (1997). Distinctions between fraud, bias, errors, misunderstanding, and incompetence. Controlled Clinical Trials, 18(6), 637 – 650.

Dorfman, P. M. & Vyverman, K.. (2009). The SAS Hash Object in Action. Proceedings of the SAS Global Forum 2009. Cary, NC: SAS Institute, Inc.

Dorfman, P. M. (2000). Private Detectives in a Data Warehouse: Key-Indexing, Bitmapping, and Hashing. Proceedings of the 25th Annual SAS Users Group International Meeting. Cary, NC: SAS Institute, Inc.

Flavin, J. M. & Carpenter, A. L. (2001). Taking Control and Keeping It: Creating and using conditionally executable SAS® Code. Proceedings of the 26th Annual SAS Users Group International Meeting. Cary, NC: SAS Institute, Inc.

Gupta, S. (2006). WHERE vs. IF Statements: Knowing the Difference in How and When to Apply. Proceedings of the 31st Annual SAS Users Group International Meeting. Cary, NC: SAS Institute, Inc.

Lafler, K.P. (2003). Undocumented and Hard-to-find SQL Features. Proceedings of the 28th Annual SAS Users Group International Meeting. Cary, NC: SAS Institute, Inc.

Lafler, K.P. (2004a). Efficiency Techniques for Beginning PROC SQL Users. Proceedings of the 29th Annual SAS Users Group International Meeting. Cary, NC: SAS Institute, Inc.

Lafler, K.P. (2004b). PROC SQL: Beyond the Basics Using SAS. Cary, NC: SAS Institute, Inc.

Lafler, K.P. (2005). Manipulating Data with PROC SQL. Proceedings of the 30th Annual SAS Users Group International Meeting. Cary, NC: SAS Institute, Inc.

Pagé, Jacques. (2004). Automated distribution of SAS results. Proceedings of the 29th Annual SAS Users Group International Meeting. Cary, NC: SAS Institute, Inc.

Scerbo, M. (2007). Win the Pennant! Use PROC FORMAT. Proceedings of the SAS Global Forum 2007. Cary, NC: SAS Institute, Inc.

Schacherer, C.W. (2012). The FILENAME Statement: Interacting with the world outside of SAS®. Proceedings of the SAS Global Forum 2012. Cary, NC: SAS Institute, Inc.

Schacherer, C.W. & Detry, M.A. (2010). PROC SQL: From Select to Pass-Through SQL. Proceedings of the South Central SAS User's Group. Cary, NC: SAS Institute, Inc.

Schacherer, C.W. & Steines, T.J. (2010). Building an Extract, Transform, and Load (ETL) Server Using Base SAS, SAS/SHARE, SAS/CONNECT, and SAS/ACCESS. Proceedings of the Midwest SAS Users Group. Cary, NC: SAS Institute, Inc.

CONTACT INFORMATION

Your comments and questions are valued and encouraged. Contact the author at:

Christopher W. Schacherer, Ph.D.
Clinical Data Management Systems, LLC
Phone: 608.478.0029
E-mail: CSchacherer@cdms-llc.com
Web: www.cdms-llc.com