SAS104-2024

Data Diagnostics with Base SAS®

Mark Jordan, SAS Institute Inc.

ABSTRACT

When you gain access to new data sources, it's important to become familiar with the data and clean and validate it before conducting analysis, reporting, and modeling. The data may include records that violate business rules, non-standard casing, spellings, or abbreviations for text data, or numeric outliers. This hands-on workshop teaches you how to use PROC SQL, PROC SGPLOT, PROC FREQ, and PROC FORMAT to generate reports useful for data profiling. You will then use PROPCASE, UPCASE, and PRX functions to standardize character data. The presentation includes an introduction to regular expressions. We'll show how to use PROC SUMMARY and the DATA step to automate identification and isolation of rows containing numeric outliers. And finally, we'll use constraints and audit trails to easily identify data that does not conform to business rules.

Data profiling

cleanme.va_rents - 23,300 rows

Column	DataType	Size	UniqueValues	MissingValues	SpecialMissingValues
ID	NCHAR	10	23297	Yes	No
county_name	NCHAR	21	10	No	No
metro_name	NCHAR	76	2	No	No
fips_code	NCHAR	10	4	No	No
statename	NCHAR	8	1	No	No
statecode	NCHAR	2	1	No	No
Bedrooms	DOUBLE	8	9	No	No
Type	NCHAR	15	5	Yes	No
Rent	DOUBLE	8	33	Yes	No

When I first gain access to new data, I run some profiling code to get a feel for what the data is like. I like to know a bit about the data types and uniqueness of the values

ID	county_name	metro_name	fips_code	statename	statecode	Bedrooms	Type	Rent
JCC0001	James City County	Virginia Beach-Norfolk-Newport News, VA-NC HUD Metro FMR Area	5109599999	Virginia	VA	4	Townhouse	999999
JCC0002	James City County	Virginia Beach-Norfolk-Newport News, VA-NC HUD Metro FMR Area	5109599999	Virginia	VA	2	Multi-Family	0.999999
JCC0003	James City County	Virginia Beach-Norfolk-Newport News, VA-NC HUD Metro FMR Area	5109599999	Virginia	VA	4	Townhouse	.

A look at a few rows of data is also useful to get the sense of what kinds of values are included in each column.

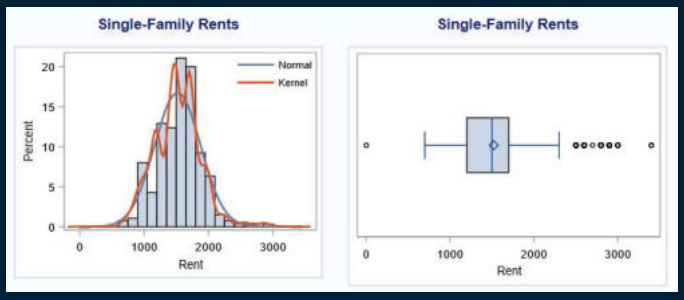
county_name	Frequency
City of Williamsburg	2090
Jaimes City County	1
James City Country	1
James City County	8
James-City County	7190
New Kent County	6500
Williamsburg County	1
Williamsburg city	8
Wmbg city	1
York County	7500

Type	Frequency
Missing	4
Apartment	4762
Efficiency	4738
Multi-family	4700
Single-family	4573
Townhouse	4522
townhouse	1

statecode	Frequency
Missing	3
VA	23294
WV	3

statename	Frequency
Virginia	23297
virginia	3

Character values are generally treated as categorical, so it's good to familiarize myself with the distribution of values. Here we can see that the county names for the City of Williamsburg and James City County have been entered several different ways. Sometimes, it's just a matter of inconsistent casing, and sometimes, some data is missing or just plain wrong. These values will need to be standardized before I can conduct any meaningful analysis.



...	Bedrooms	Type	Rent
...	4	Townhouse	1750
...	2	Single-Family	1250
...	4	Townhouse	999999

For numerical data, I like to take a look at the numerical distribution, too. For example, single-family rents seem to be fairly rationally distributed, with a few outliers here and there, but there's something seriously wrong with those townhouse rent values!

Standardization

```
Type=propcase(type);
StateName=propcase(StateName);
StateCode='VA';
```

Type	Frequency
Missing	4
Apartment	4762
Efficiency	4738
Multi-family	4700
Single-family	4573
Townhouse	4522
townhouse	1

statecode	Frequency
Missing	3
VA	23294
WV	3

statename	Frequency
Virginia	23297
virginia	3

Before we get any deeper into analyzing numerical data in categories, we'll need to standardize the category values. Things like normalizing capitalization and replacing easily correctable bad values go a long way towards simplifying the rest of the process.

```
if upcase(county_name)in ('CITY OF WILLIAMSBURG'
                          ,'WILLIAMSBURG COUNTY'
                          ,'WILLIAMSBURG CITY'
                          ,'WMBG CITY')
   then county_name='City of Williamsburg';
else if
   upcase(county_name)in ('JAMES CITY COUNTY'
                          ,'JAMES CITY COUNTRY'
                          ,'JAIMES CITY COUNTY'
                          ,'JAMES-CITY COUNTY')
   then county_name='James City County';
```

county_name
City of Williamsburg
Jaimes City County
James City Country
James City County
James-City County
New Kent County
Williamsburg County
Williamsburg city
Wmbg city
York County

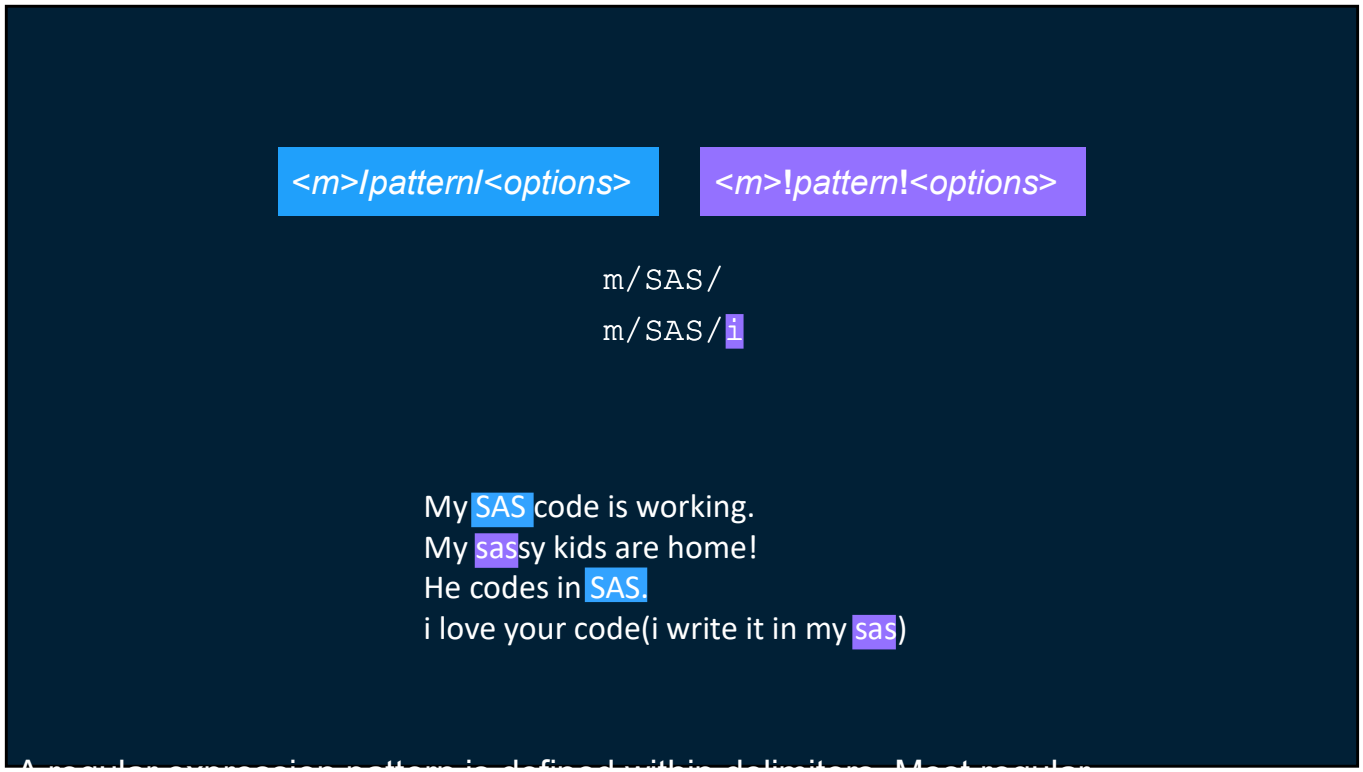
Standardizing text values can be accomplished using standard logical techniques, but the code can get quite complex.

Regular Expressions to the Rescue!

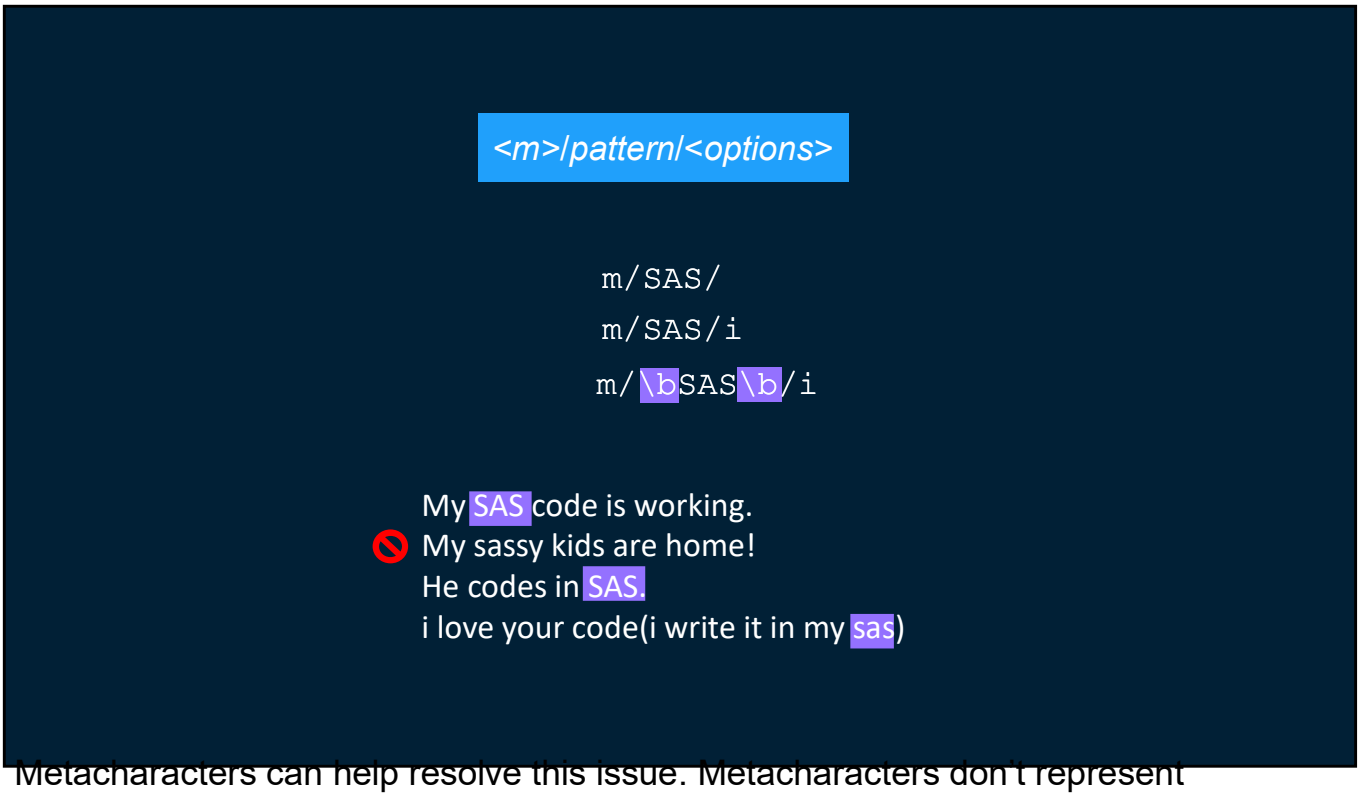
Regular expressions

- Describe a character pattern that must be matched for a line of text to be selected
- Patterns are described using plain text and metacharacters
- Metacharacters match character types or classes instead of exact matches

Using regular expressions can make the code much more manageable. Instead of requiring you to explicitly list every value you're looking for, reguar expresions let you specify a pattern of text using a mixture of literal characters and metacharacters which represent a class or set of values insted of a single specific value.



This match expression merely matches all cases of SAS in upper case. Comparisons are case sensitive, but the "I" option can be specified to ignore case. However, now those sassy kids are also selected…



Metacharacters can help resolve this issue. Metacharacters don't represent actual text – they represent a class of characters instead. For example, \b represents a WORD boundary. This expression selects the letters s a s no matter what the case, but only when the letters are preceded and followed by a word boundary. This excludes those sassy kids.

Selected Metacharacters

Meta-character	Matches
\b	Any word boundary
\d	Any digit
\s	Any white space character (space, tab, return, form feed, etc.)
\w	Any word character (letter, digit, or underscore)
\W	Any non-word character (not a letter, digit, or underscore)
\	Escape character: the next character is a literal, not a metacharacter
^	The beginning of the text string
$	The end of the text string

Metacharacters provide a rich an succint lexicon for describing text patterns. Note that in a regular expression, CASE MATTERS! For example, \w represents any word character, and \W any non-word character. That's the general rule – the upper case version describes the opposite of the lower case class. These metacaracters all include the back slash, but there are several metacharacters that are just a single character. To refer to the actual character instead of the metacharacter, we'll use a back slash as an escape character.

Meta-character	Description
(<*pattern*>)	Specifies a group identified by a sub-expression
(:?<*pattern*>)	Specifies a non-capturing group identified by a sub-expression
*	Matches the preceding sub-expression zero or more times
+	Matches the preceding sub-expression one or more times
?	Matches the preceding sub-expression zero or one time
{n}	Matches the preceding sub-expression exactly n times
{n,}	Matches the preceding sub-expression at least n times
{n,m}	Matches the preceding sub-expression at least n and at most m times

You can use parentheses to work with a sub-expression as a group, and multipliers to specify how many times a sub-expression pattern should be matched.

Metacharacters as Text

Example:
Find all of "A+" grades

Mark's Grade: B-
Jane's Grade: A+
Jerry's Grade: A
Susan's Grade: A+

```
/Grade: A+/
```

In this example, we are searching the text for A+ grades. In regular

Example:
Find all of "A+" grades

Mark's Grade: B-
Jane's Grade: A+
Jerry's Grade: A
Susan's Grade: A+

```
/Grade: A\+/
```

Adding the backslash escape character to the pattern matches and returns plus

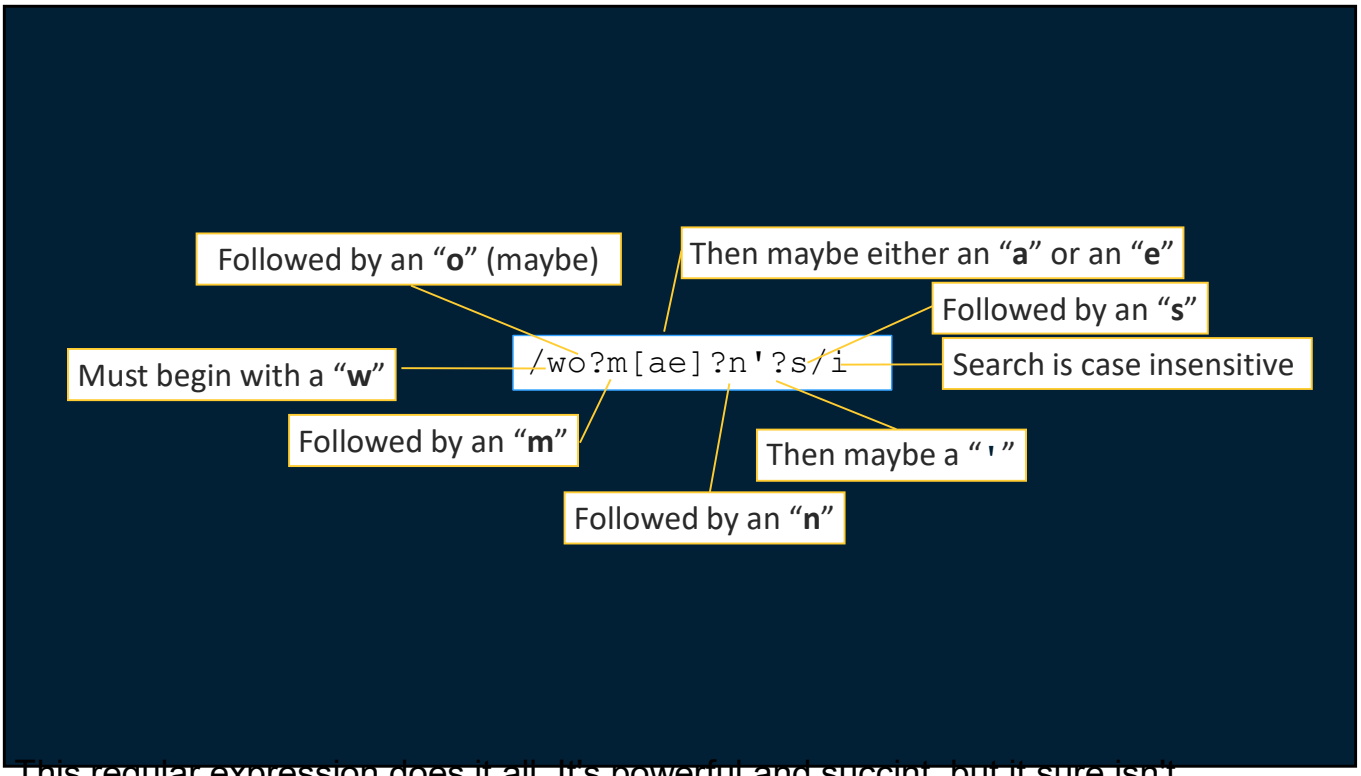
Known pattern variations:

- Woman's
- Women's
- Womns
- Wmens
- Wmns

```
if find(text,"Woman's")
   or find(text,"Women's")
   or find(text,"Womns")
   or find(text,"Wmens")
   or find(text,"Wmns");
```

But what if case was also a problem?

Say you have some data containing women's items. The data has been entered in a myraid of different ways. We could use FIND and conditional logic to write the code, but the more varaitions that appear, the logner the code becomes. And what if casing is allso inconsistent?



This regular expression does it all. It's powerful and succint, but it sure isn't intuitive for humans to read! Let's break that down. First, it specifies that the pattern must begin with the letter w, which might or might not be followed by a single letter o. The next letter must be an m, which might be followed by either an a or an e. Next, an n is required, which might be followed by an apostrophe, and finally an s is required. The option 'i' makes the search case-insensitive.

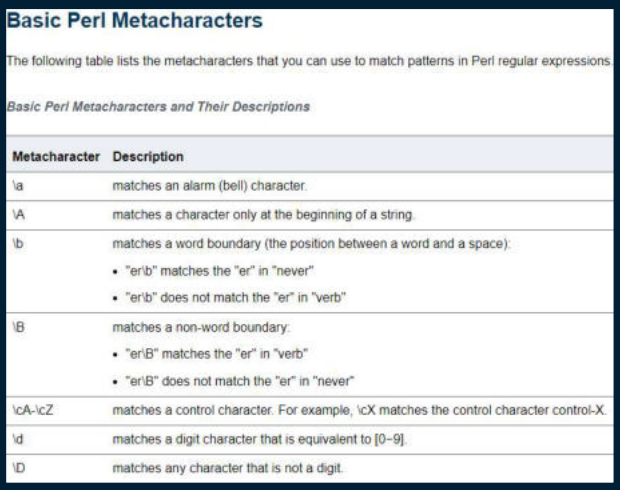
Regular Expression Groups

```
/* match either Side A or Side B*/
m/Side (A|B)/i
```

```
/* match any text containing exactly 3 words */
m/^(\w+\W+){3}$/
```

Parentheses in regular expressions indicate grouping and are useful for logical operations.

SAS Documentation



Basic Perl Metacharacters

The following table lists the metacharacters that you can use to match patterns in Perl regular expressions.

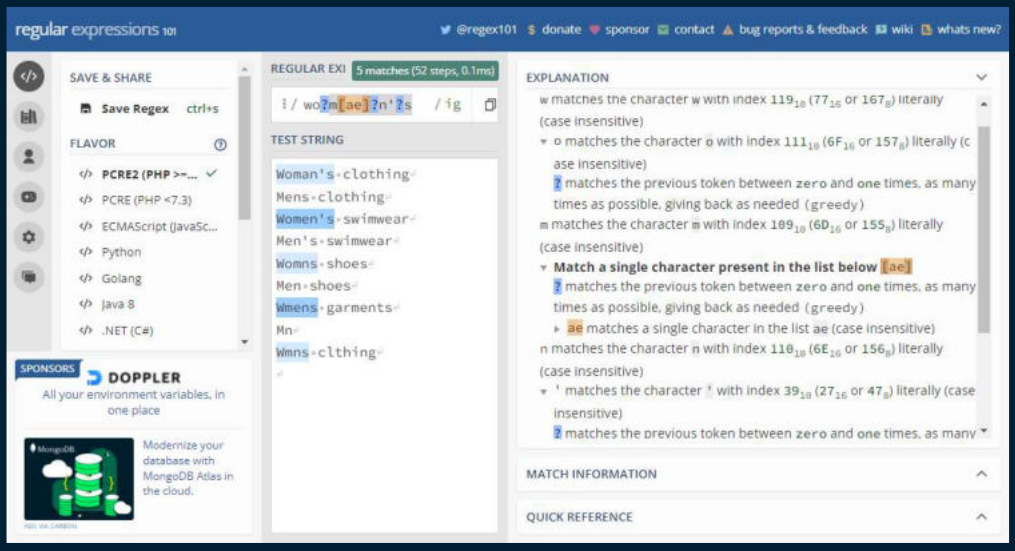
Basic Perl Metacharacters and Their Descriptions

Metacharacter	Description
\a	matches an alarm (bell) character.
\A	matches a character only at the beginning of a string.
\b	matches a word boundary (the position between a word and a space): • "er\b" matches the "er" in "never" • "er\b" does not match the "er" in "verb"
\B	matches a non-word boundary: • "er\B" matches the "er" in "verb" • "er\B" does not match the "er" in "never"
\cA-\cZ	matches a control character. For example, \cX matches the control character control-X.
\d	matches a digit character that is equivalent to [0-9].
\D	matches any character that is not a digit.

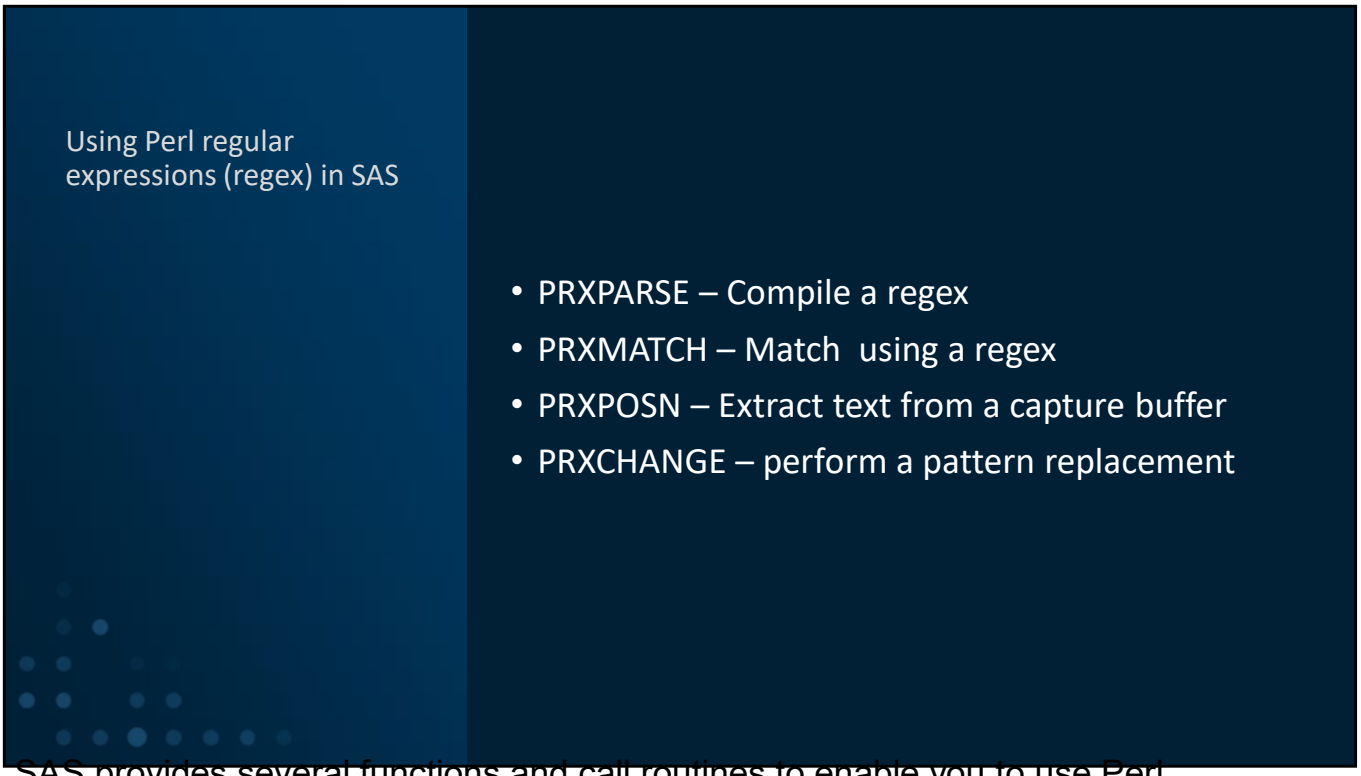
The SAS documentation includes a comprehensive list of Perl regular expression metacharacters

(see https://go.documentation.sas.com/doc/en/pgmsascdc/default/lefunctionsref/p0s9ilagexmjl8n1u7e1t1jfnzlk.htm?fromDefault=)

Testing regular expressions at regex101.com



You can test your regular expressions or decipher inherited regular expressions at https://regex101.com.



SAS provides several functions and call routines to enable you to use Perl regular expressions in your SAS code. The names all begin with “PRX”, so I refer to them as the PRX functions.

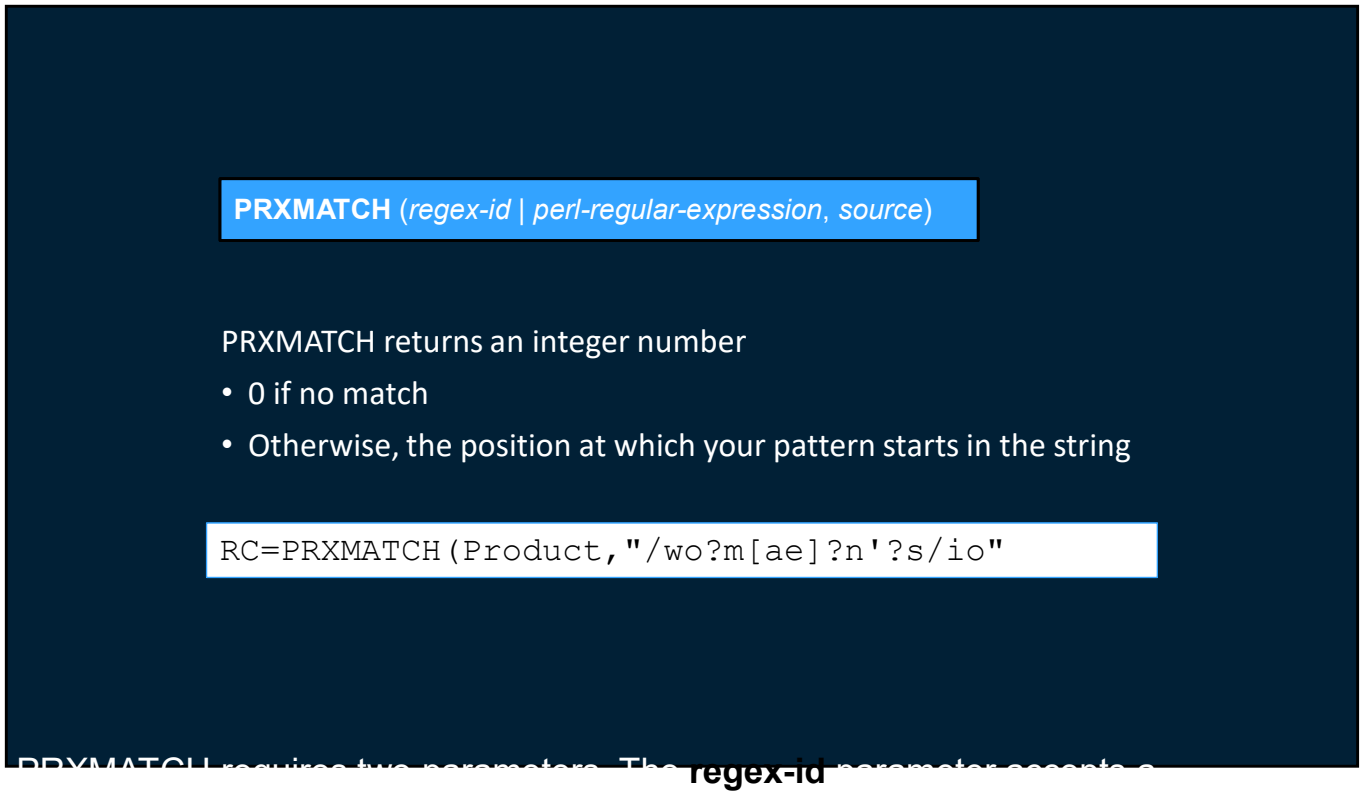
PRXPARSE (*perl-regular-expression*)

PRXPARSE returns an integer

- A regular-expression-id if the expression compiled
- A missing value if there was an error

```
ERROR: Closing delimiter "/" not found after regular expression "/xxx".
ERROR: The regular expression passed to the function PRXPARSE contains
       a syntax error.
```

PRXPARSE accepts a single parameter consisting of a valid Perl regular expression expressed as a string. PRXPARSE returns an integer. If the expression compiled correctly, the integer serves as a ID that can be used in place of the expression in other PRX functions. Compiling a regular expression is relativley resource intensive, so we usually capture and retain the ID so we execute PRXPARSE only once in the DATA step. If the expression doesn't compile, a zero is returned and explanatory error message is generated.



PRXMATCH requires two parameters. The **regex-id** parameter accepts a

an valid

Perl regular expression expressed as a string

options follow the closing slash of the expression. The 'i' makes the search case-insensitive, and the 'o' ensures that the expression compiles only once.

PRXCHANGE(*regex-id*, *times*, *source*)

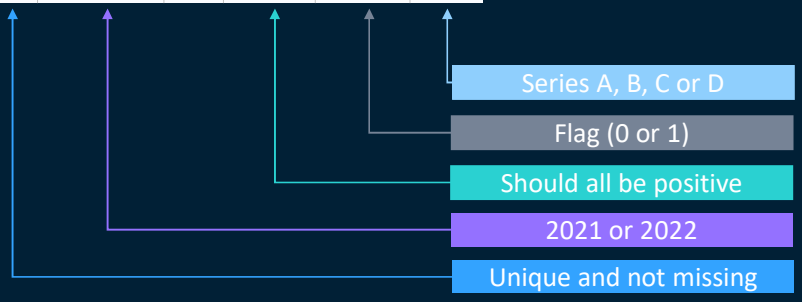
PRXCHANGE returns a character value

- The entire modified text from the source
- If pattern is not found, the source text is not modified

PRXCHANGE performs a pattern-matching text substitution and requires three parameters. As before, the **regex-id** parameter accepts a valid Perl regular expression. The **times** parameter specifies the number of times to search for and replace the matching pattern, and t Specifying –1 for the **times** parameter causes matching and replacing to repeat until the end of the **source** text is reached. >PRXCHANGE returns a character string containing the entire value of **source** with the specified changes applied. If there is no match, PRXCHANGE returns the original **source** value unchanged.

Diagnostics: PROC FREQ for categorical and missing values

ID	Date	Type	Number	Selected	Mixed
0001	02/24/2022	d	2.27605	1	D001
0002	01/27/2022	c	2.25073	0	C002
0003	02/12/2022	d	2.75687	1	D003
0004	03/29/2022	a	1.51051	0	A004
0005	01/10/2022	c	2.66001	0	C005



Categorical variable diagnostics generally involve identifying if the categories are valid according to some specified criteria. For example, an ID may be required to be unique and not missing, dates may be required to fall within a specific range, numeric values may have certain constraints, and validation may involve just a portion of a text variable.

```
proc format;
   value num low-<0='Bad'
         other='OK'
         .='OK'
   ;
   value $cmiss (default=20) " " ='Missing'
   ;
run;

proc freq data=cleanme.categories;
   table date type number selected mixed/nocum missing;
   format date year4. type $cmiss. number num. mixed $1. ;
run;
```

Because PROC FREQ can work with formatted values, a well-designed custom format can make it easier to identify problems in the data. The **num** format flags values below zero as Bad, and all others, including missing values, as Good. The **cmiss** character format flags missing character values with the word 'Missing'. When we apply the formats in PROC FREQ, don't forget to include missing values in the report.

Type	Frequency	Percent
Missing	95	19.00
A	95	19.00
B	93	18.60
C	100	20.00
D	117	23.40

Mixed	Frequency	Percent
A	190	38.00
B	93	18.60
C	100	20.00
D	117	23.40

Date	Frequency	Percent
2022	500	100.00

Number	Frequency	Percent
Bad	13	2.60
OK	487	97.40

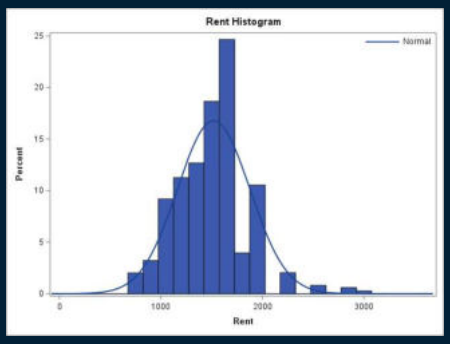
Selected	Frequency	Percent
0	257	51.40
1	243	48.60

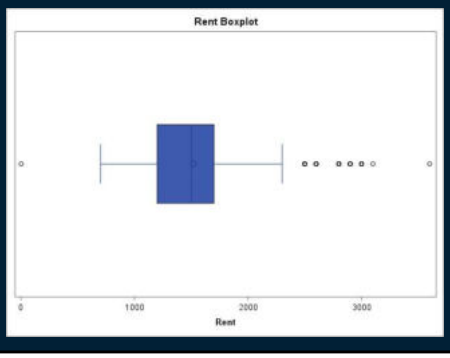
Because we specified the MISSING option on the tables statement, the report for the TYPE variable includes missing values in the body of the report, and the percentages reported include the missing values. About 1/5 of all Type values are missing – an important thing to know about our data. Our **num** format treats missing values as OK. Applying the format to the Number column made it very easy to evaluate the state of the data in that column.

Numeric Anomalies – Interquartile Range

```
title "Rent Histogram";
proc sgplot data=cleanme.va_rent;
  where Type="Single-Family";
  histogram Rent;
  density Rent;
run;

title "Rent Boxplot";
proc sgplot data=cleanme.va_rent;
  where Type="Single-Family";
  hbox Rent;
run;
```

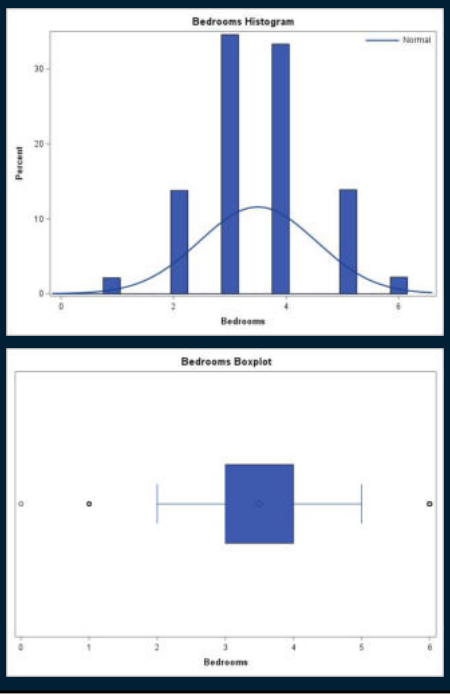




Box plots and histograms provide an easy way to visualize outliers.

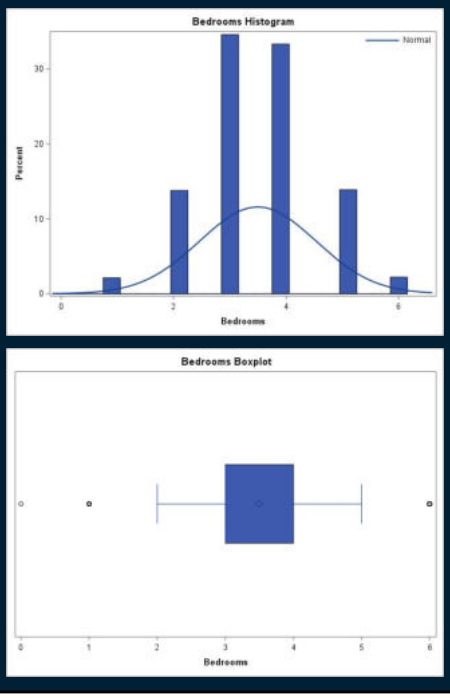
```
title "Bedrooms Histogram";
proc sgplot data=cleanme.va_rent;
  where Type="Townhouse";
  histogram Rent;
  density Rent;
run;

title " Bedrooms Boxplot";
proc sgplot data=cleanme.va_rent;
  where Type="Townhouse";
  hbox Rent;
run;
```



But I'd have to write the code for every variant of every analysis variable.

```
%rentplots(cleanme.va_rent)
```



I'd rather use a macro for repetitive work. Even so, I'm going to have to LOOK at each of these graphs and the write corrective code. If I could only get this information as data instead of as a graphic…

PROC SUMMARY for Data-Driven Diagnostics

```
proc summary data=cleanme.va_rent;
   var bedrooms;
   class type;
   output out=work.va_rent_stats
      Min= Q1= Median= QRange= Q4= Max= / autoname;
run;
```

Type	Bedrooms_Min	Bedrooms_Q1	Bedrooms_Median	Bedrooms_QRange	Bedrooms_Q3	Bedrooms_Max
	0.000	1	3	3	4	999
Apartment	1.000	2	3	1	3	999
Efficiency	0.000	0	0	0	0	0
Multi-Family	0.999	2	2	1	3	6
Single-Family	0.999	3	4	1	4	999
Townhouse	0.000	3	3	1	4	6

Good news! You can use PROC SUMMARY to generate the information necessary to diagnose outliers programmatically. The upper and lower hinge ("whisker") values of a box plot can be calculated using the Q1, Qrange, and Q3 values. You can then use those hinge values to programmatically identify and isolate outliers in your data.

```
if _N_ = 1 then do;
   declare hash iqr(dataset:'va_rent_stats');
   iqr.defineKey('Type');
   iqr.defineData('Rent_Q1','Bedrooms_Q1','Rent_Q3'
         ,'Bedrooms_Q3','Rent_QRange','Bedrooms_QRange');
   iqr.defineDone();
end;
set cleanme.va_rent;
/* Compare rent to high/low cutoffs */
if iqr.find()=0 then do;
   if Rent < Rent_Q1-1.5*Rent_QRange then
      Reason = CATX(', ',Reason,'Low Rent');
   else if Rent > Rent_Q3+1.5*Rent_QRange then
      Reason = CATX(', ',Reason,'High Rent');
…
```

Because no sorting or merging is required, I like to use hash objects to look up the limits for each row of the original data. The find method looks up the upper and lower values for the record and we can then test to see if this record is an outlier.

Integrity Constraints and Audit Trails

- CHECK - Limits data values based on a user-defined constraint.
- NOT NULL - Disallows missing (null) values.
- UNIQUE - Requires a specified variable to be unique, even if it is null
- PRIMARY KEY - Requires the value to be both UNIQUE and NOT NULL.

Integrity constraints can be applied to tables to enforce business rules when data is added. We'll find these constraints most useful:
The CHECK constraints limits data values based on a user-defined specification.
The NOT NULL constraints prevents entering missing (null) values in a column.
The UNIQUE constraints requires each value in this column to be unique. A single null value would pass this test.
The PRIMARY KEY constraints requires the value to be both UNIQUE and NOT NULL.

When an audit trail is applied to a SAS dataset, an additional dataset is created to track all changes made. The process is handled transparently by the SAS system.

ID	statename	statecode	Bedrooms	Rent	Type
Required, unique	'Virginia'	'VA"	Between 0 and 6	Between 500 and 5,500	Must be one of: • Efficiency • Single-Family • Townhouse • Multi-Family • Apartment
PRIMARY KEY	CHECK	CHECK	CHECK	CHECK	CHECK

To meet the business rules specified for the column, most columns will require a CHECK constraint, but the ID column requires PRIMARY KEY instead.

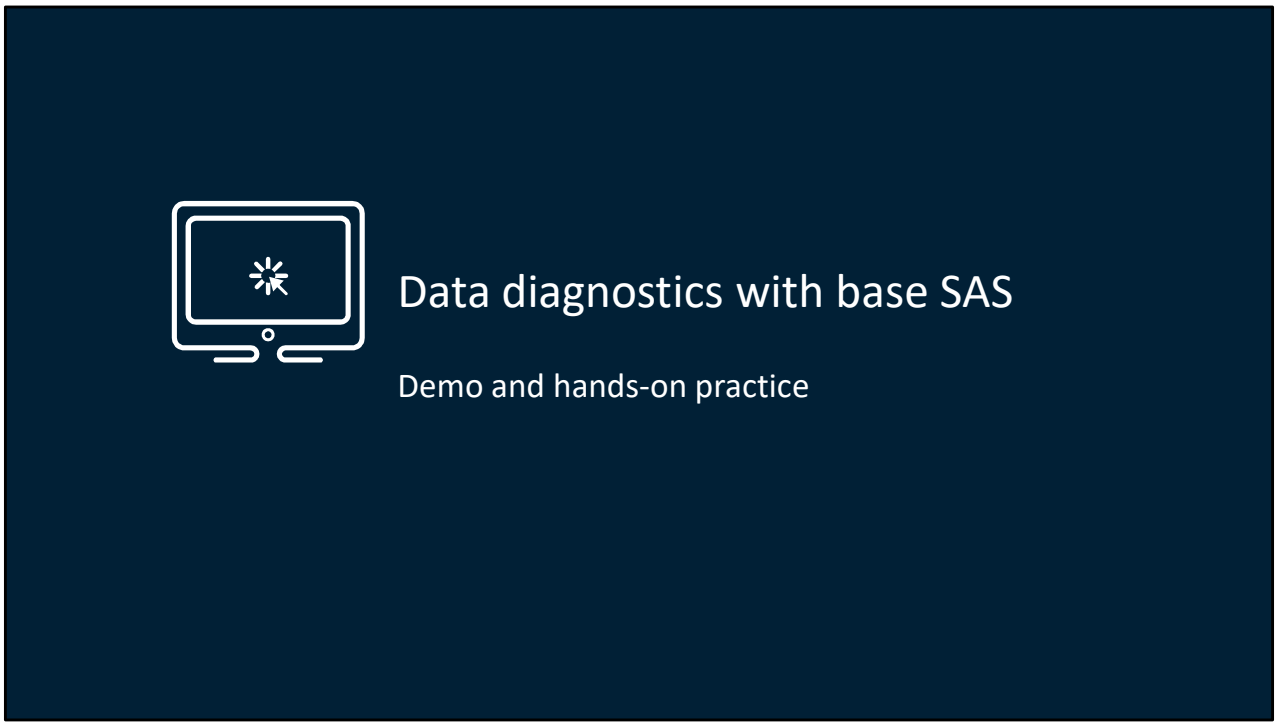
```
proc datasets library=work nolist nodetails;
   modify va_rent;
   /* Primary key ensures the value is not NULL and is unique */
   ic create ID_pk = primary key (ID)
      message="ID missing or not unique" msgtype=user;
   /* The where= expression describes a GOOD row. Others are rejected. */
   ic create StateCode_ck = check (where=(StateCode='VA'))
      message="State code not VA" msgtype=user;
   ic create StateName_ck = check (where=(StateName='Virginia'))
      message="State name not Virginia" msgtype=user;
   ic create Type_ck = check
      (where=(Type in('Efficiency','Single-Family','Townhouse','Multi-Family','Apartment')))
       message="Improper TYPE" msgtype=user;
   ic create TypeBR_ck = check
      (where=((Bedrooms<1 and Type='Efficiency') or Bedrooms>=1))
       message="Type-Bedroom mismatch" msgtype=user;
   ic create BR_ck = check
      (where=(0<=bedrooms<=6 and int(Bedrooms)=bedrooms))
       message="Check bedroom count" msgtype=user;
run;quit;
```

We can use PROC DATASETS to modify an existing datasets and apply integrity constraints. However, if any of the existing data doesn't meet the requirements, the constraint can't be applied. To use integrity constraints for diagnostics, we first create an empty dataset with the same structure as the original, apply the integrity constraints to the empty table.

```
proc datasets library=work nolist nodetails;
   audit va_rent;
   initiate;
run;quit;
```

```
proc print data=work.va_rent(type=audit);
run;
```

With integrity constraints in effect, we can use PROC DATASETS to initiate an audit trail. Then, appending the original data to the empty table will cause only "clean" records to be accepted, and the audit records can be used to identify every record that was rejected. The audit trail data can be accessed using the type=audit dataset option.



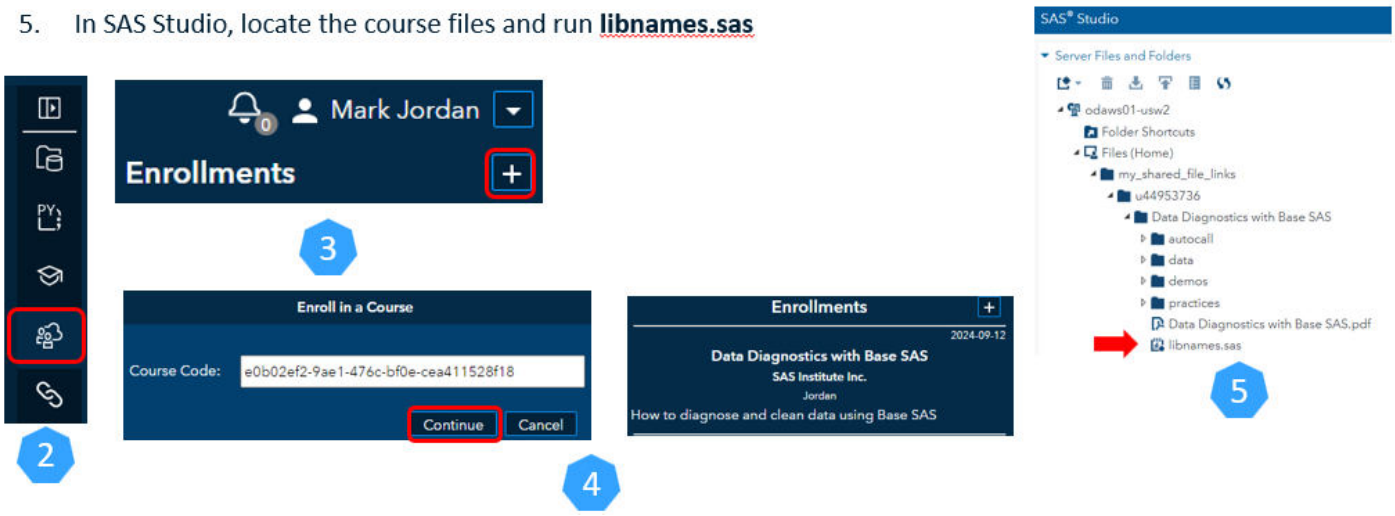
I'll demo using base SAS tools for data diagnostics, then you can try your hand with the practices

Set up for practices. Choose the instructions for your choice of SAS environment:

SAS on Demand for Academics **Course Code is:**
e0b02ef2-9ae1-476c-bf0e-cea411528f18

Practice – Setup in SAS on Demand for Academics (SoDA)

1. Log in to SAS on Demand for Academics.
2. In the right navigation bar, select Enrollments.
3. In Enrollments, click the plus sign to add a course.
4. Enter the course code and click **Continue** to add the course to your enrolments.
5. In SAS Studio, locate the course files and run **libnames.sas**



To use your own SAS:

Practice – Setup in Your Own SAS Installation:

1. Open SAS Studio.
2. In the left navigation bar **Server Files and Folders**, locate **Files (Home).**
3. Right click **Files (Home)** and create a *new folder* named **HOW.**
4. Press F4 to open a new program tab. Enter and run the following code:

```
filename setup url https://bit.ly/SASJediDDsetup;
%include setup;
%SETUPDATADIAGNOSICS(~/HOW)
```

5. Results:

Row	Library	Table	Obs Count	File Size (MB)
1	CLEANME	FORMATS	5	0.3
2	CLEANME	VA_RENT	23,300	3.9
3	CLEANME	VA_RENT_DEMO	16,393	2.9
4	CLEANME	VA_RENT_PRACTICE	6,907	1.3

This workshop was developed using SAS on Demand for Academics – and if you haven't already, you should register for your free account right now! It's a great place to practice and learn to code in SAS.

To grab the programs and data for this presentation, download the zip file from https://bit.ly/SASJediDiagnostics. After unzipping the programs and data to a location on your hard drive, upload them with the same folder structure to a location accessible in your SAS session. Then run the **setup.sas** program found in the folder named **data_diagnostics** to set up your system, and you're ready to run the demo code yourself and work on the practices.

Contact Information

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

Mark Jordan, SAS Jedi, Retired
Email: sas.jedi@gmail.com
X: @SASJedi