SAS - Only Important Parts

Chitranshu Vashishth

29th January 2025

Table of Contents

3. Date Shifting (Past and Future Dates)
4. Date Components and Extraction
5. Complex Date Calculations for Credit Risk
6. Working with Time Components
7. Date/Time Function Reference Table
8. Common Date Formats

Best Practices:

Sorting a dataset and De-duplicating a dataset on different set of columns or entire dataset

1. Basic Sorting Using PROC SORT
2. De-duplication Using NODUPKEY/NODUPREC
3. Comprehensive Example with Credit Data
4. Advanced De-duplication Using DATA Step
5. Using FIRST. and LAST. Variables
6. Using SQL for De-duplication
7. Complex Sorting and De-duplication
8. Best Practices and Tips

Common Pitfalls to Avoid:

Macro Programming

1. Macro Variables - Basic Creation and Usage
2. Referencing Macro Variables
3. Macro Functions
4. Complex Macro Example for Credit Risk Analysis
5. Advanced Macro Techniques
6. Error Handling and Debugging
7. Utility Macros

Best Practices:

Creating custom macro functions

1. Basic Macro Function Structure
2. Simple Custom Functions
3. String Manipulation Functions
4. Date Manipulation Functions
5. Validation Functions
6. Complex Business Logic Functions
7. Utility Functions
8. Error Handling Functions

Best Practices:

Loops within data step and within Macros

1. DATA Step Loops
    A. DO Loop (Simple Counter)
    B. DO WHILE Loop
    C. DO UNTIL Loop
    D. Nested DO Loops
    E. Complex DATA Step Loop Example
2. Macro Loops
    A. %DO Loop
    B. %DO %WHILE Loop
    C. %DO %UNTIL Loop
    D. Complex Macro Loop Examples
3. Combined DATA Step and Macro Loops
4. Error Handling in Loops

Best Practices:

Create Datasets and set statement.

Here's a comprehensive tutorial on creating datasets and using the SET statement in SAS:

1. Creating Datasets in SAS

Basic Explanation:

In SAS, datasets (also called SAS tables) are the primary way to store and manipulate data. There are multiple ways to create datasets:

- Using DATA step
- Using PROC SQL
- Importing from external sources

Key Methods:

1. **Basic Dataset Creation**

```
DATA new_dataset;
    input name $ age salary;
    datalines;
John 30 50000
Mary 25 45000
Steve 35 60000
;
run;
```

2. **Creating Dataset from Existing Dataset**

```
DATA new_dataset;
    SET existing_dataset;
run;
```

3. **Creating Empty Dataset**

```
/* Method 1: Using NULL dataset */
DATA empty_dataset;
    SET sashelp.class(obs=0);
run;

/* Method 2: Using STOP statement */
DATA empty_dataset;
    stop;
run;

/* Method 3: With defined structure */
DATA empty_dataset;
    length name $50 age 8 salary 8;
```

```
    stop;
run;
```

2. SET Statement

Basic Explanation:

The SET statement is used to read observations from one or more existing SAS datasets.

Key Uses:

1. **Reading Single Dataset**

```
DATA work.new_data;
    SET library.original_data;
run;
```

2. **Combining Multiple Datasets (Vertically)**

```
DATA combined_data;
    SET dataset1 dataset2 dataset3;
run;
```

3. **Using SET with Options**

```
DATA filtered_data;
    SET original_data(keep=account_id balance status);
    WHERE balance > 1000;
run;
```

Practical Example for Credit Risk Context:

```
/* Creating a customer credit dataset */
DATA work.customer_credit;
    input customer_id name $ credit_score payment_status $;
    datalines;
1001 John 720 Good
1002 Mary 680 Fair
1003 Steve 800 Excellent
;
run;

/* Creating an empty template for bad loans */
```

```
DATA work.bad_loans;
    length customer_id 8
           name $50
           credit_score 8
           payment_status $10
           default_date 8;
    format default_date date9.;
    stop;
run;

/* Combining customer data with additional filters */
DATA work.high_risk_customers;
    SET work.customer_credit;
    WHERE credit_score < 700;
    risk_flag = 'High Risk';
run;
```

Common Pitfalls to Avoid:

1. Not specifying variable lengths in empty datasets
2. Forgetting to use RUN statements
3. Not considering dataset options (like KEEP, DROP, WHERE) when using SET
4. Not handling missing values appropriately

Best Practices:

1. Always define variable attributes (length, format) explicitly for empty datasets
2. Use meaningful dataset names
3. Comment your code
4. Consider using WHERE clauses for filtering during SET operations
5. Always verify the dataset creation with PROC CONTENTS or PROC PRINT

Keep columns when reading / appending / manipulating

KEEP Statement and Options

Basic Explanation:

KEEP is used to select specific variables (columns) to retain in the output dataset. It can be used in multiple ways:

1. As a dataset option
2. As a statement in DATA step
3. During dataset manipulation operations

Different Methods to Use KEEP

1. KEEP as Dataset Option

```
/* Using KEEP as dataset option */
DATA new_dataset;
    SET original_dataset(KEEP=customer_id account_balance credit_score);
run;
```

2. KEEP as Statement

```
/* Using KEEP as statement */
DATA new_dataset;
    SET original_dataset;
    KEEP customer_id account_balance credit_score;
run;
```

3. KEEP in Multiple Dataset Scenarios

```
/* When combining datasets */
DATA combined_data;
    SET dataset1(KEEP=id name balance)
        dataset2(KEEP=id name balance);
run;
```

Practical Example for Credit Risk Context:

```
/* First, create sample credit data */
DATA credit_master;
    input customer_id name $ age address $
          credit_score income account_balance
          payment_history $ phone $;
    datalines;
1001 John 30 NYC 750 60000 5000 Good 555-0101
1002 Mary 25 LA 680 45000 2000 Fair 555-0102
1003 Steve 35 CHI 800 80000 10000 Excellent 555-0103
;
run;

/* Example 1: Keep essential credit risk variables */
DATA credit_risk_analysis;
    SET credit_master(KEEP=customer_id credit_score
                           income account_balance
                           payment_history);
run;

/* Example 2: Multiple datasets with different KEEP options */
DATA demographics credit_info;
    SET credit_master;
    /* Demographic information */
    IF _N_=1 THEN DO;
        KEEP customer_id name age address;
        OUTPUT demographics;
    END;
    /* Credit information */
    IF _N_=1 THEN DO;
        KEEP customer_id credit_score income payment_history;
        OUTPUT credit_info;
    END;
run;

/* Example 3: Combining with WHERE clause */
DATA high_risk_customers;
    SET credit_master(KEEP=customer_id credit_score
                           payment_history income);
    WHERE credit_score < 700 AND income < 50000;
run;
```

Advanced Usage with MERGE:

```
/* Using KEEP with MERGE statement */
DATA combined_risk_profile;
    MERGE customer_base(KEEP=customer_id name credit_score)
          transaction_history(KEEP=customer_id total_spend frequency);
    BY customer_id;
run;
```

Important Considerations:

1. **Performance Impact**:

- Using KEEP as a dataset option is more efficient than using it as a statement
- It reduces I/O by reading only necessary variables

2. **Timing Differences**:

```
/* These two are not equivalent */

/* Option 1: Variables dropped before processing */
DATA new_data;
    SET old_data(KEEP=var1 var2);
    var3 = var1 + var2;
run;

/* Option 2: Variables dropped after processing */
DATA new_data;
    SET old_data;
    var3 = var1 + var2;
    KEEP var1 var2;
run;
```

Best Practices:

1. **Documentation**

```
/* Always document which variables are being kept and why */
DATA risk_metrics /* Contains core risk variables */;
    SET full_customer_data(KEEP=
        customer_id    /* Primary key */
        credit_score   /* Risk indicator */
        payment_status /* Current status */
    );
run;
```

2. **Variable Lists**

```
/* Using variable lists with KEEP */
DATA analysis_ready;
    SET raw_data(KEEP=
        customer_id
        name
        credit_: /* Keeps all variables starting with 'credit_' */
        risk_: /* Keeps all variables starting with 'risk_' */
    );
run;
```

Common Pitfalls to Avoid:

1. Not checking if required variables exist before using KEEP
2. Forgetting that KEEP options are processed before statements
3. Not considering the order of operations when using multiple data steps
4. Dropping variables needed for calculations before they're used

Quick Reference Table:

Method	Usage	When to Use
Dataset Option	dataset(KEEP=var1 var2)	Most efficient, use when possible
Statement	KEEP var1 var2;	When need to keep variables based on conditions
Multiple Datasets	SET data1(KEEP=...) data2(KEEP=...);	When combining datasets

Drop columns when reading / appending / manipulating

DROP Statement and Options

Basic Explanation:

DROP is the opposite of KEEP - it's used to remove specific variables (columns) from the output dataset. Like KEEP, it can be used in multiple ways:

1. As a dataset option
2. As a statement in DATA step
3. During dataset manipulation operations

Different Methods to Use DROP

1. DROP as Dataset Option

```
/* Using DROP as dataset option */
DATA new_dataset;
    SET original_dataset(DROP=phone_number address zip_code);
run;
```

2. DROP as Statement

```
/* Using DROP as statement */
DATA new_dataset;
    SET original_dataset;
    DROP phone_number address zip_code;
run;
```

3. DROP in Multiple Dataset Scenarios

```
/* When combining datasets */
DATA combined_data;
    SET dataset1(DROP=unnecessary_var1)
        dataset2(DROP=unnecessary_var2);
run;
```

Practical Example for Credit Risk Context:

```
/* First, create sample credit data */
DATA credit_master;
    input customer_id name $ age address $
          credit_score income account_balance
          payment_history $ phone $ email $
          marketing_flag $ last_contact_date $;
    datalines;
1001 John 30 NYC 750 60000 5000 Good 555-0101 john@email.com Y 01JAN2024
1002 Mary 25 LA 680 45000 2000 Fair 555-0102 mary@email.com N 15JAN2024
1003 Steve 35 CHI 800 80000 10000 Excellent 555-0103 steve@email.com Y
20JAN2024
;
run;

/* Example 1: Drop contact information for analysis */
DATA credit_analysis;
    SET credit_master(DROP=phone email marketing_flag
                          last_contact_date);
run;

/* Example 2: Multiple datasets with different DROP options */
DATA risk_profile customer_contact;
    SET credit_master;
    /* Risk Profile - Drop contact info */
    IF _N_=1 THEN DO;
        DROP phone email address;
        OUTPUT risk_profile;
    END;
    /* Contact Info - Drop financial info */
    IF _N_=1 THEN DO;
        DROP credit_score income account_balance;
        OUTPUT customer_contact;
    END;
run;

/* Example 3: Combining with WHERE clause */
DATA high_balance_customers;
    SET credit_master(DROP=marketing_flag last_contact_date);
    WHERE account_balance > 5000;
run;
```

Advanced Usage:

1. Using DROP with MERGE:

```
/* Using DROP with MERGE statement */
DATA risk_report;
    MERGE customer_data(DROP=internal_id update_date)
          credit_history(DROP=process_date batch_id);
    BY customer_id;
run;
```

2. Using DROP with Variable Lists:

```
/* Dropping variables with common prefixes */
DATA clean_data;
    SET raw_data(DROP=temp_: /* Drops all variables starting with
'temp_' */
                     int_:  /* Drops all variables starting with 'int_'
*/);
run;
```

Important Considerations:

1. **Performance Impact**:

```
/* More efficient - drops variables before reading */
DATA new_data;
    SET old_data(DROP=var1-var10);
run;

/* Less efficient - reads all variables first */
DATA new_data;
    SET old_data;
    DROP var1-var10;
run;
```

2. **Timing Differences**:

```
/* These produce different results */

/* Option 1: Variables dropped before processing */
DATA new_data;
    SET old_data(DROP=var1);
    total = var1 + var2; /* This will fail - var1 already dropped */
```

```
run;

/* Option 2: Variables dropped after processing */
DATA new_data;
    SET old_data;
    total = var1 + var2;
    DROP var1;
run;
```

Best Practices:

1. **Documentation**

```
/* Document dropped variables */
DATA risk_analysis /* Core risk analysis dataset */;
    SET full_data(DROP=
        temp_var1    /* Temporary calculation var */
        batch_id     /* Processing variable */
        update_flag  /* System flag */
    );
run;
```

2. **Efficient Variable Selection**

```
/* Drop multiple related variables efficiently */
DATA clean_customer_data;
    SET raw_customer_data(DROP=
        _temp: /* Temporary variables */
        _flag: /* Flag variables */
        _sys_: /* System variables */
    );
run;
```

Common Pitfalls to Avoid:

1. Dropping variables needed for calculations
2. Not considering the order of operations
3. Dropping key identifier variables
4. Using DROP when KEEP might be more appropriate

Quick Reference Table:

Method	Usage	Best Use Case
Dataset Option	dataset(DROP=var1 var2)	Most efficient, use when possible
Statement	DROP var1 var2;	When need to drop after calculations
Variable Lists	DROP var: temp:	When dropping groups of similarly named variables

Additional Tips:

1. Use PROC CONTENTS to verify dropped variables:

```
PROC CONTENTS DATA=new_dataset;
run;
```

2. Combine with other options:

```
DATA new_dataset;
    SET original_dataset(DROP=var1-var5
                         WHERE=(balance > 1000)
                         FIRSTOBS=100);
run;
```

Data manipulation and creating new columns

Data Manipulation and Column Creation

Basic Explanation:

In SAS, you can create new columns through:

1. Direct assignment
2. Mathematical operations

3. Conditional logic
4. Functions
5. String operations

1. Basic Column Creation

```
DATA new_data;
    SET original_data;

    /* Simple assignment */
    new_column = existing_column;

    /* Mathematical operation */
    total_balance = checking_balance + savings_balance;

    /* Percentage calculation */
    utilization_ratio = (current_balance / credit_limit) * 100;
run;
```

2. Practical Credit Risk Example

```
/* Create sample credit data */
DATA credit_data;
    input customer_id
          credit_score
          income
          debt_amount
          months_on_book
          current_balance;
    datalines;
1001 720 60000 15000 24 2000
1002 680 45000 20000 36 5000
1003 800 80000 10000 12 1000
;
run;

/* Comprehensive data manipulation example */
DATA credit_analysis;
    SET credit_data;

    /* 1. Basic calculations */
    debt_to_income = (debt_amount / income) * 100;
    monthly_income = income / 12;

    /* 2. Conditional logic using IF-THEN */
    IF credit_score >= 750 THEN credit_rating = 'Excellent';
```

```
    ELSE IF credit_score >= 700 THEN credit_rating = 'Good';
    ELSE IF credit_score >= 650 THEN credit_rating = 'Fair';
    ELSE credit_rating = 'Poor';

    /* 3. Using SELECT-WHEN */
    SELECT;
        WHEN (months_on_book >= 36) tenure_segment = 'Long-term';
        WHEN (months_on_book >= 12) tenure_segment = 'Medium-term';
        OTHERWISE tenure_segment = 'New-customer';
    END;

    /* 4. Creating flags */
    high_risk_flag = (credit_score < 650 AND debt_to_income > 40);

    /* 5. Using SAS functions */
    review_date = TODAY();
    months_since_review = INTCK('month', review_date, TODAY());

    /* 6. Numeric transformations */
    log_income = LOG(income);
    income_bracket = ROUND(income, 10000);

    /* 7. Creating categorical variables */
    LENGTH income_category $20;
    IF income < 30000 THEN income_category = 'Low';
    ELSE IF income < 70000 THEN income_category = 'Medium';
    ELSE income_category = 'High';

    /* 8. Complex calculations */
    risk_score = (credit_score * 0.4) +
                 ((1 - debt_to_income/100) * 30) +
                 (months_on_book * 0.5);
run;
```

3. String Manipulation Examples

```
DATA string_manipulations;
    SET customer_data;

    /* Converting case */
    upper_name = UPCASE(name);
    lower_name = LOWCASE(name);
    proper_name = PROPCASE(name);

    /* Extracting substrings */
    first_char = SUBSTR(name, 1, 1);

    /* Combining strings */
```

```
    full_address = CATX(' ', address, city, state, zip);

    /* Removing spaces */
    clean_id = COMPRESS(account_id);
run;
```

4. Date Manipulations

```
DATA date_calculations;
    SET account_data;

    /* Create date variables */
    today = TODAY();
    current_time = TIME();

    /* Calculate durations */
    account_age_days = TODAY() - account_open_date;
    account_age_years = YRDIF(account_open_date, TODAY(), 'AGE');

    /* Format dates */
    FORMAT today account_open_date date9.
           current_time time8.
           account_age_years 5.2;
run;
```

5. Advanced Calculations

```
DATA risk_metrics;
    SET loan_data;

    /* Moving averages */
    IF _N_ >= 3 THEN DO;
        moving_avg_3m = MEAN(OF payment_amt{1:3});
    END;

    /* Retain variables across observations */
    RETAIN previous_balance 0;
    balance_change = current_balance - previous_balance;
    previous_balance = current_balance;

    /* Arrays for multiple calculations */
    ARRAY payments{12} payment1-payment12;
    total_payments = SUM(OF payments{*});
    avg_payment = MEAN(OF payments{*});
run;
```

Common Functions for Credit Risk Analysis:

```
DATA risk_analysis;
    SET credit_data;

    /* Statistical functions */
    payment_std = STD(OF payment1-payment12);
    payment_max = MAX(OF payment1-payment12);
    payment_min = MIN(OF payment1-payment12);

    /* Probability functions */
    normal_score = PROBNORM((credit_score - 700)/100);

    /* Rounding functions */
    rounded_score = ROUND(risk_score, 0.5);
    ceiling_value = CEIL(calculated_risk);
    floor_value = FLOOR(calculated_risk);
run;
```

Best Practices:

1. Always initialize variables:

```
LENGTH new_var $50;  /* For character variables */
new_numeric = 0;     /* For numeric variables */
```

2. Use meaningful variable names
3. Document calculations
4. Consider missing values:

```
IF NOT MISSING(income) AND NOT MISSING(debt) THEN
    debt_ratio = debt / income;
```

Common Pitfalls to Avoid:

1. Not handling missing values
2. Not declaring variable lengths for character variables
3. Not considering the order of operations
4. Forgetting to format new variables
5. Not checking for division by zero

Rename a column in-place or otherwise

Renaming Columns in SAS

There are several methods to rename columns in SAS:

1. Using RENAME= dataset option
2. Using RENAME statement
3. Using PROC DATASETS
4. Using PROC SQL

1. Using RENAME= Dataset Option

```
/* Basic syntax */
DATA new_dataset;
    SET original_dataset(RENAME=(old_name=new_name));
run;

/* Practical example */
DATA credit_analysis;
    SET raw_data(RENAME=(
        cust_id=customer_id
        cr_score=credit_score
        bal=balance
    ));
run;
```

2. Using RENAME Statement

```
/* Basic syntax */
DATA new_dataset;
    SET original_dataset;
    RENAME old_name=new_name;
run;

/* Practical example */
DATA risk_profile;
    SET customer_data;
    RENAME customer_number=customer_id
```

```
            risk_points=risk_score;
run;

/* Renaming inplace by creating new column, dropping older column,
renaming this new column */
DATA risk_profile;
    SET customer_data;
    new_date = old_date
    DROP old_date
    RENAME new_date = old_date_on_boarding
run;
```

3. Using PROC DATASETS (In-Place Renaming)

```
/* Basic syntax */
PROC DATASETS LIBRARY=libname NOLIST;
    MODIFY dataset_name;
    RENAME old_name=new_name;
QUIT;

/* Practical example */
PROC DATASETS LIBRARY=WORK NOLIST;
    MODIFY credit_data;
    RENAME customer_num=customer_id
           credit_pts=credit_score
           account_bal=balance;
QUIT;
```

4. Using PROC SQL

```
/* Basic syntax */
PROC SQL;
    CREATE TABLE new_dataset AS
    SELECT old_name AS new_name
    FROM original_dataset;
QUIT;

/* Practical example */
PROC SQL;
    CREATE TABLE credit_analysis AS
    SELECT customer_num AS customer_id,
           credit_pts AS credit_score,
           account_bal AS balance,
           other_columns /* Other columns remain unchanged */
    FROM raw_credit_data;
QUIT;
```

Practical Examples in Credit Risk Context:

1. Comprehensive Example with Multiple Renaming Methods:

```
/* Create sample data */
DATA credit_raw;
    input cust_num score bal status $ dt;
    format dt date9.;
    datalines;
1001 720 5000 A 21500
1002 680 3000 B 21500
1003 800 7000 A 21500
;
run;

/* Method 1: Using RENAME= option */
DATA credit_clean;
    SET credit_raw(RENAME=(
        cust_num=customer_id
        score=credit_score
        bal=balance
        dt=review_date
    ));
run;

/* Method 2: Using RENAME statement */
DATA credit_clean2;
    SET credit_raw;
    RENAME cust_num=customer_id
           score=credit_score
           bal=balance
           dt=review_date;
run;

/* Method 3: In-place renaming */
PROC DATASETS LIBRARY=WORK NOLIST;
    MODIFY credit_raw;
    RENAME cust_num=customer_id
           score=credit_score
           bal=balance
           dt=review_date;
QUIT;

/* Method 4: Using PROC SQL */
PROC SQL;
    CREATE TABLE credit_clean3 AS
    SELECT cust_num AS customer_id,
           score AS credit_score,
           bal AS balance,
```

```
            status,
            dt AS review_date
    FROM credit_raw;
QUIT;
```

2. Renaming Multiple Variables with Pattern:

```
/* For variables with similar patterns */
DATA credit_history;
    SET monthly_data(RENAME=(
        bal1=balance_month1
        bal2=balance_month2
        bal3=balance_month3
        pay1=payment_month1
        pay2=payment_month2
        pay3=payment_month3
    ));
run;
```

3. Combining with Other Operations:

```
/* Renaming while selecting specific variables */
DATA risk_analysis;
    SET customer_master(
        RENAME=(
            cust_id=customer_id
            risk_sc=risk_score
        )
        KEEP=cust_id risk_sc balance status
    );
run;
```

Best Practices:

1. **Consistent Naming Convention**

```
/* Use consistent naming patterns */
DATA consistent_names;
    SET raw_data(RENAME=(
        cust_id=customer_id        /* Use full words */
        fname=first_name           /* Avoid abbreviations */
        lname=last_name
        ph_num=phone_number
    ));
run;
```

2. **Documentation**

```
/* Document rename operations */
DATA credit_clean;
    SET credit_raw(RENAME=(
        cust_num=customer_id    /* Primary identifier */
        scr=credit_score        /* Standardized score name */
        dt=review_date          /* Consistent date naming */
    ));
    /* Add rename explanation */
    ATTRIB customer_id LABEL="Unique Customer Identifier"
           credit_score LABEL="Credit Score (300-850 range)"
           review_date LABEL="Latest Review Date";
run;
```

Common Pitfalls to Avoid:

1. Not checking if new names already exist
2. Using reserved words as new names
3. Not maintaining proper variable lengths after renaming
4. Forgetting to rename related variables consistently

Quick Reference Table:

Method	Use Case	Advantages	Disadvantages
RENAME= Option	During data step processing	Clear syntax	Creates new dataset
RENAME Statement	Simple renaming	Easy to read	Creates new dataset

Method	Use Case	Advantages	Disadvantages
PROC DATASETS	In-place renaming	No new dataset needed	Must be careful with active datasets
PROC SQL	Complex transformations	Flexible	May be slower for simple renaming

---

Change formatting of a column for existing dataset, or while at creation/appending/merge

- **Text to Number**
- **Number to Text**
- **A string/number denoting date in one format to another string/number denoting that date in other format**
- **Text to Date**
- **Number to Date**
- **Date to Text**
- **Date to Number**

Format Conversions in SAS

Basic Methods to Change Formats:

1. Using FORMAT statement
2. Using INPUT/PUT functions
3. Using ATTRIB statement
4. Using PROC DATASETS

1. Text to Number Conversions

```
DATA conversion_example;
    /* Sample string data */
    input text_value $20.;
    datalines;
1234
5678.90
-123.45
;

    /* Method 1: Using INPUT function */
    number1 = INPUT(text_value, 8.);

    /* Method 2: Using INPUT with specific format */
    number2 = INPUT(text_value, COMMA10.2);

    /* Method 3: Best practice with error handling */
    if _ERROR_=0 then
        number3 = INPUT(COMPRESS(text_value), BEST12.);
run;
```

2. Number to Text Conversions

```
DATA string_conversion;
    /* Sample numeric data */
    input number 8.;
    datalines;
1234
5678.90
-123.45
;

    /* Method 1: Using PUT function */
    text_value1 = PUT(number, 8.);

    /* Method 2: Using PUT with specific format */
    text_value2 = PUT(number, DOLLAR10.2);

    /* Method 3: With leading zeros */
    text_value3 = PUT(number, Z6.);
run;
```

3. Date Format Conversions

```
DATA date_conversions;
    /* Various date inputs */
    input date_str $10. number_date 8.;
    datalines;
01JAN2024 20240101
15FEB2024 20240215
31DEC2023 20231231
;

    /* String to Date */
    date1 = INPUT(date_str, DATE9.);

    /* Number to Date */
    date2 = INPUT(PUT(number_date, 8.), YYMMDD8.);

    /* Date to String */
    date_text1 = PUT(date1, DATE9.);
    date_text2 = PUT(date1, DDMMYY10.);

    /* Date to Number */
    date_num = INPUT(PUT(date1, DATE9.), YYMMDD8.);

    /* Format specifications */
    FORMAT date1 date2 DATE9.
           date_num YYMMDD10.;
run;
```

4. Comprehensive Credit Risk Example

```
/* Create sample credit data with mixed formats */
DATA credit_raw;
    input customer_id $
          credit_score $
          review_date $
          balance $
          status $;
    datalines;
A12345 720 20240115 5000.00 ACTIVE
B67890 680 20240120 3500.50 REVIEW
C11111 800 20240125 7500.75 ACTIVE
;
run;

/* Comprehensive format conversion */
DATA credit_clean;
```

```
    SET credit_raw;

    /* Convert credit score from string to numeric */
    credit_score_num = INPUT(credit_score, 8.);

    /* Convert balance from string to numeric */
    balance_num = INPUT(balance, COMMA10.2);

    /* Convert date string to SAS date */
    review_date_sas = INPUT(review_date, YYMMDD8.);

    /* Create formatted strings */
    formatted_balance = PUT(balance_num, DOLLAR12.2);
    formatted_date = PUT(review_date_sas, DATE9.);

    /* Format specifications */
    FORMAT review_date_sas DATE9.
           balance_num DOLLAR12.2
           credit_score_num 8.;
run;
```

5. Changing Formats for Existing Dataset

```
/* Method 1: Using PROC DATASETS */
PROC DATASETS LIBRARY=WORK NOLIST;
    MODIFY credit_clean;
    FORMAT balance_num DOLLAR12.2
           review_date_sas MMDDYY10.
           credit_score_num 4.;
QUIT;

/* Method 2: Using DATA step */
DATA credit_clean;
    SET credit_clean;
    FORMAT balance_num DOLLAR12.2
           review_date_sas MMDDYY10.
           credit_score_num 4.;
run;
```

6. Common Date Format Conversions

```
DATA date_examples;
    /* Various date format scenarios */

    /* String dates in different formats */
    date_str1 = '01JAN2024';
```

```
    date_str2 = '2024-01-01';
    date_str3 = '01/01/2024';

    /* Convert to SAS dates */
    sas_date1 = INPUT(date_str1, DATE9.);
    sas_date2 = INPUT(date_str2, YYMMDD10.);
    sas_date3 = INPUT(date_str3, MMDDYY10.);

    /* Convert to different formats */
    formatted_date1 = PUT(sas_date1, DATE9.);
    formatted_date2 = PUT(sas_date1, DDMMYY10.);
    formatted_date3 = PUT(sas_date1, YYMMDD10.);
    formatted_date4 = PUT(sas_date1, WEEKDATE.);

    /* Numeric representation */
    date_num = INPUT(PUT(sas_date1, DATE9.), YYMMDD8.);

    FORMAT sas_date1-sas_date3 DATE9.;
run;
```

Common Format Types:

```
/* Numeric Formats */
FORMAT numeric_var
    BEST12.        /* Best representation */
    COMMA10.2      /* With commas and decimals */
    DOLLAR12.2     /* Currency format */
    PERCENT8.2     /* Percentage format */
    Z6.            /* Leading zeros */
;

/* Date Formats */
FORMAT date_var
    DATE9.         /* Default date format */
    DATETIME20.    /* Date and time */
    DDMMYY10.      /* DD/MM/YYYY */
    MMDDYY10.      /* MM/DD/YYYY */
    YYMMDD10.      /* YYYY-MM-DD */
    WEEKDATE.      /* Full week date */
;
```

Best Practices:

1. Always validate conversions:

```
/* Validation example */
DATA _NULL_;
    converted_value = INPUT('123.45', 8.);
    IF _ERROR_ THEN PUT 'ERROR: Invalid conversion';
run;
```

2. Handle missing values:

```
DATA clean_data;
    SET raw_data;
    IF NOT MISSING(string_value) THEN
        numeric_value = INPUT(string_value, 8.);
run;
```

3. Document format changes:

```
/* Add labels and formats */
PROC DATASETS LIBRARY=WORK NOLIST;
    MODIFY dataset_name;
    LABEL variable1='Description of variable1'
          variable2='Description of variable2';
    FORMAT variable1 FORMAT1.
           variable2 FORMAT2.;
QUIT;
```

Formats for string, number, date in SAS. How is date stored as and how to handle it

1. String (Character) Formats

```
/* Basic string formats */
FORMAT
    name $12.          /* Fixed width character */
    address $UPCASE.   /* Uppercase */
    city $PROPCASE.    /* Proper case */
    state $2.          /* Two-character state code */
;

/* Example usage */
DATA string_formats;
    input text $20.;
    datalines;
JOHN SMITH
new york city
123 Main St
;

    /* Different string formats */
    name_upper = PUT(text, $UPCASE12.);
    name_proper = PUT(text, $PROPCASE20.);
    name_left = PUT(text, $LEFT12.);
    name_right = PUT(text, $RIGHT12.);
run;
```

2. Numeric Formats

```
/* Basic numeric formats */
FORMAT
    amount DOLLAR12.2    /* $1,234.56 */
    rate PERCENT8.2      /* 12.34% */
    score BEST12.        /* Best representation */
    id Z6.               /* Leading zeros */
    decimal 8.2          /* Fixed decimal */
    comma COMMA10.2      /* With thousands separator */
;
```

```
/* Example usage */
DATA numeric_formats;
    input value;
    datalines;
1234.56
0.1234
123456
;

    /* Different numeric formats */
    FORMAT
        value_dollar DOLLAR12.2
        value_percent PERCENT8.2
        value_comma COMMA10.2
        value_scientific E12.
        value_binary BINARY8.
    ;
run;
```

3. Date Formats and Storage

How SAS Stores Dates:

- SAS stores dates as numbers representing days since January 1, 1960
- Negative numbers represent dates before 1960
- Time is stored as number of seconds since midnight
- Datetime is stored as seconds since January 1, 1960

```
/* Understanding date storage */
DATA date_storage;
    /* Create various date values */
    date1 = '01JAN2024'd;  /* Date constant */
    date2 = TODAY();       /* Current date */
    time1 = TIME();        /* Current time */
    dt1 = DATETIME();      /* Current datetime */

    /* Show raw numeric values */
    raw_date = date1;      /* Days since 1960 */
    raw_time = time1;      /* Seconds since midnight */
    raw_dt = dt1;          /* Seconds since 1960 */

    /* Format specifications */
    FORMAT
        date1 date2 DATE9.
        time1 TIME8.
        dt1 DATETIME20.
        raw_date raw_time raw_dt BEST12.
```

```
    ;
run;
```

4. Common Date Formats

```
/* Comprehensive date format example */
DATA date_formats;
    date_value = '01JAN2024'd;

    /* Different date formats */
    FORMAT
        date1 DATE9.        /* 01JAN2024 */
        date2 DDMMYY10.     /* 01/01/2024 */
        date3 MMDDYY10.     /* 01/01/2024 */
        date4 YYMMDD10.     /* 2024-01-01 */
        date5 WEEKDATE.     /* Monday, January 1, 2024 */
        date6 MONYY7.       /* JAN2024 */
        date7 YEAR4.        /* 2024 */
        date8 WORDDATX.     /* January 1, 2024 */
    ;

    /* Assign same date with different formats */
    date1 = date_value;
    date2 = date_value;
    date3 = date_value;
    date4 = date_value;
    date5 = date_value;
    date6 = date_value;
    date7 = date_value;
    date8 = date_value;
run;
```

5. Date Handling and Calculations

```
/* Date manipulation examples */
DATA date_handling;
    /* Create some dates */
    start_date = '01JAN2024'd;
    end_date = '31DEC2024'd;

    /* Basic calculations */
    days_between = end_date - start_date;
    years_between = YRDIF(start_date, end_date, 'AGE');
    months_between = INTCK('MONTH', start_date, end_date);

    /* Add intervals */
```

```
    next_month = INTNX('MONTH', start_date, 1);
    next_quarter = INTNX('QTR', start_date, 1);
    next_year = INTNX('YEAR', start_date, 1);

    /* Beginning and end of periods */
    month_start = INTNX('MONTH', start_date, 0, 'B');
    month_end = INTNX('MONTH', start_date, 0, 'E');

    /* Extract components */
    year = YEAR(start_date);
    month = MONTH(start_date);
    day = DAY(start_date);

    /* Format specifications */
    FORMAT start_date end_date
           next_month next_quarter next_year
           month_start month_end DATE9.;
run;
```

6. Datetime and Time Formats

```
/* Time and datetime handling */
DATA time_formats;
    current_dt = DATETIME();
    current_time = TIME();

    /* Different time formats */
    FORMAT
        time1 TIME8.         /* 14:30:00 */
        time2 TIMEAMPM8.     /* 2:30 PM */
        time3 HHMM5.         /* 14:30 */

        dt1 DATETIME20.      /* 01JAN2024:14:30:00 */
        dt2 DTDATE9.         /* 01JAN2024 */
        dt3 DTWKDATX.        /* Monday, January 1, 2024 */
    ;

    time1 = current_time;
    time2 = current_time;
    time3 = current_time;

    dt1 = current_dt;
    dt2 = current_dt;
    dt3 = current_dt;
run;
```

7. Best Practices for Date Handling

```
/* Date handling best practices */
DATA date_best_practices;
    /* 1. Always validate date inputs */
    input_date = '2024-01-01';
    valid_date = INPUT(input_date, YYMMDD10.);
    IF NOT MISSING(valid_date);

    /* 2. Handle missing dates */
    IF MISSING(valid_date) THEN valid_date = .;

    /* 3. Use date constants carefully */
    fixed_date = '01JAN2024'd;

    /* 4. Use relative dates */
    today = TODAY();
    yesterday = TODAY() - 1;
    last_month = INTNX('MONTH', TODAY(), -1);

    /* 5. Standard format for storage */
    FORMAT valid_date fixed_date
           today yesterday last_month DATE9.;
run;
```

Common Date Functions Reference:

Function	Purpose	Example
TODAY()	Current date	today = TODAY();
MDY()	Create date from M/D/Y	date = MDY(1,1,2024);
YEAR()	Extract year	year = YEAR(date);
MONTH()	Extract month	month = MONTH(date);
DAY()	Extract day	day = DAY(date);
WEEKDAY()	Get day of week	dow = WEEKDAY(date);
INTCK()	Count intervals	months = INTCK('MONTH', date1, date2);
INTNX()	Advance intervals	next_month = INTNX('MONTH', date, 1);

Function	Purpose	Example
YRDIF()	Calculate years	years = YRDIF(date1, date2, 'AGE');

Length, Format, Informat

1. LENGTH Statement

Used to specify the number of bytes for storing variables.

```
/* Basic LENGTH syntax */
DATA example;
    LENGTH
        customer_id 8       /* Numeric, 8 bytes */
        name $50           /* Character, 50 bytes */
        address $100       /* Character, 100 bytes */
        balance 8          /* Numeric, 8 bytes */
    ;
run;
```

2. FORMAT Statement

Specifies how values are displayed when printed or viewed.

```
/* Basic FORMAT syntax */
DATA example;
    FORMAT
        balance DOLLAR12.2    /* $1,234.56 */
        rate PERCENT8.2       /* 12.34% */
        date DATE9.           /* 01JAN2024 */
        name $50.             /* Character format */
    ;
run;
```

3. INFORMAT Statement

Specifies how SAS should read input values.

```
/* Basic INFORMAT syntax */
DATA example;
    INFORMAT
        date YYMMDD10.        /* Reads 2024-01-01 */
        balance DOLLAR12.2    /* Reads $1,234.56 */
        text $50.             /* Reads character */
    ;
run;
```

Comprehensive Example Using All Three

```
/* Create sample credit risk data */
DATA credit_data;
    /* Define lengths */
    LENGTH
        customer_id $10    /* Character ID */
        name $50          /* Customer name */
        status $10        /* Account status */
        notes $200        /* Comments field */
        credit_score 8    /* Numeric score */
        balance 8         /* Account balance */
        review_date 8     /* Date field */
    ;

    /* Define formats for display */
    FORMAT
        credit_score BEST12.
        balance DOLLAR12.2
        review_date DATE9.
        status $UPCASE.
    ;

    /* Define informats for reading */
    INFORMAT
        credit_score BEST12.
        balance COMMA12.2
        review_date YYMMDD10.
        status $UPCASE.
    ;

    /* Sample data */
    input customer_id $ name $ credit_score balance review_date
YYMMDD10.
          status $;
    datalines;
C001 John_Smith 720 5000.50 2024-01-15 active
C002 Mary_Jones 680 3500.75 2024-01-16 review
C003 Bob_Wilson 800 7500.25 2024-01-17 active
```

```
;
run;
```

Common Format Types Reference Table

Category	Format	Example	Description
Numeric	BEST12.	1234.56	Best representation
	COMMA12.2	1,234.56	With commas
	DOLLAR12.2	$1,234.56	Currency
	PERCENT8.2	12.34%	Percentage
	Z6.	001234	Leading zeros
Character	$CHAR.	Text	Standard character
	$UPCASE.	TEXT	Uppercase
	$PROPCASE.	Text	Proper case
Date	DATE9.	01JAN2024	Standard date
	DDMMYY10.	01/01/2024	DD/MM/YYYY
	DATETIME20.	01JAN2024:12:00:00	Date and time

Advanced Usage Examples

1. Complex Data Structure

```
/* Creating a complex credit risk dataset */
DATA risk_analysis;
    /* Length definitions */
    LENGTH
        account_id $15        /* Account identifier */
        customer_name $50     /* Full name */
        risk_category $10     /* Risk classification */
        review_notes $500     /* Detailed notes */
        credit_score 8        /* Credit score */
        default_prob 8        /* Default probability */
        last_review 8         /* Review date */
```

```
        next_review 8         /* Next review date */
    ;

    /* Format specifications */
    FORMAT
        credit_score BEST12.
        default_prob PERCENT8.2
        last_review DDMMYY10.
        next_review DDMMYY10.
        risk_category $UPCASE.
    ;

    /* Informat specifications */
    INFORMAT
        credit_score BEST12.
        default_prob PERCENT8.
        last_review YYMMDD10.
        next_review YYMMDD10.
        risk_category $UPCASE.
    ;
run;
```

2. Using ATTRIB Statement (Combines LENGTH, FORMAT, and INFORMAT)

```
DATA credit_profile;
    /* ATTRIB combines LENGTH, FORMAT, and INFORMAT */
    ATTRIB
        customer_id   LENGTH=$10
                      LABEL='Customer Identifier'

        credit_score LENGTH=8
                      FORMAT=BEST12.
                      LABEL='Credit Score (300-850)'

        balance       LENGTH=8
                      FORMAT=DOLLAR12.2
                      LABEL='Current Balance'

        review_date   LENGTH=8
                      FORMAT=DATE9.
                      INFORMAT=YYMMDD10.
                      LABEL='Last Review Date'

        status        LENGTH=$10
                      FORMAT=$UPCASE.
                      LABEL='Account Status'
    ;
```

```
run;
```

3. Handling Special Cases

```
DATA special_cases;
    /* Length for efficiency */
    LENGTH
        long_text $32767    /* Maximum length for character */
        small_num 3         /* Small numeric (3 bytes) */
        precise_num 8       /* Full precision (8 bytes) */
    ;

    /* Special formats */
    FORMAT
        small_num Z5.       /* Leading zeros */
        precise_num 16.8    /* High precision */
        percent_val PERCENT8.1  /* One decimal percentage */
    ;

    /* Special informats */
    INFORMAT
        date_var ANYDTDTE.  /* Flexible date reading */
        num_var COMMA12.    /* Reads numbers with commas */
    ;
run;
```

Best Practices

1. **Define Lengths Early**

```
/* Define lengths at the start of DATA step */
DATA customer_data;
    LENGTH customer_id $10 name $50 balance 8;
    /* Rest of the code */
run;
```

2. **Use Consistent Formats**

```
/* Standardize formats across program */
PROC FORMAT;
    VALUE $risk_fmt
        'H' = 'High Risk'
        'M' = 'Medium Risk'
        'L' = 'Low Risk';
run;

DATA risk_data;
    FORMAT risk_level $risk_fmt.;
    /* Rest of the code */
run;
```

3. **Document with Labels**

```
DATA documented_data;
    ATTRIB
        customer_id LENGTH=$10 LABEL='Unique Customer ID'
        balance LENGTH=8 FORMAT=DOLLAR12.2
                LABEL='Current Account Balance'
    ;
run;
```

Put, Input, Putn, Inputn and their usage within macro function context

1. Basic Functions Overview

```
/* Sample data for demonstrations */
DATA sample;
    /* Numeric values */
    num_val = 12345.67;

    /* Character values */
    char_val = '12345.67';

    /* Date value */
    date_val = '01JAN2024'd;
```

```
    /* Basic conversions */
    /* Numeric to Character */
    char_from_num = PUT(num_val, 8.2);

    /* Character to Numeric */
    num_from_char = INPUT(char_val, 8.2);

    /* Numeric to Numeric with format */
    num_formatted = PUTN(num_val, 'DOLLAR12.2');

    /* Character to Numeric with format */
    num_from_char2 = INPUTN(char_val, 'COMMA12.2');
run;
```

2. PUT vs PUTN

```
DATA put_examples;
    num_value = 12345.67;
    date_value = '01JAN2024'd;

    /* PUT - returns character */
    char_result1 = PUT(num_value, 12.2);        /* '12345.67' */
    char_result2 = PUT(num_value, DOLLAR12.2);  /* '$12,345.67' */
    char_result3 = PUT(date_value, DATE9.);     /* '01JAN2024' */

    /* PUTN - returns numeric */
    num_result1 = PUTN(num_value, '12.2');
    num_result2 = PUTN(date_value, 'DATE9.');

    /* Verify results */
    PUT "Character Results:";
    PUT char_result1= char_result2= char_result3=;
    PUT "Numeric Results:";
    PUT num_result1= num_result2=;
run;
```

3. INPUT vs INPUTN

```
DATA input_examples;
    char_value = '12345.67';
    date_str = '01JAN2024';

    /* INPUT - flexible, can return character or numeric */
    num_result1 = INPUT(char_value, 12.2);
    date_result = INPUT(date_str, DATE9.);
```

```
    /* INPUTN - always returns numeric */
    num_result2 = INPUTN(char_value, '12.2');
    date_result2 = INPUTN(date_str, 'DATE9.');

    /* Verify results */
    PUT "Results from INPUT:";
    PUT num_result1= date_result=;
    PUT "Results from INPUTN:";
    PUT num_result2= date_result2=;
run;
```

4. Macro Context Usage

```
/* Macro variable creation and manipulation */
%macro format_conversions;
    /* Create some macro variables */
    %let numeric_value = 12345.67;
    %let char_value = 12345.67;
    %let date_value = 01JAN2024;

    /* Using PUT in macro */
    %let formatted_num = %sysfunc(PUT(&numeric_value, DOLLAR12.2));

    /* Using PUTN in macro */
    %let num_formatted = %sysfunc(PUTN(&numeric_value, DOLLAR12.2));

    /* Using INPUT in macro */
    %let parsed_num = %sysfunc(INPUT(&char_value, 12.2));

    /* Using INPUTN in macro */
    %let parsed_date = %sysfunc(INPUTN(&date_value, DATE9.));

    /* Display results */
    %put Formatted Number (PUT): &formatted_num;
    %put Numeric Formatted (PUTN): &num_formatted;
    %put Parsed Number (INPUT): &parsed_num;
    %put Parsed Date (INPUTN): &parsed_date;
%mend;

%format_conversions;
```

5. Complex Examples in Credit Risk Context

```
/* Credit risk scoring with macro processing */
%macro process_credit_data;
    /* Sample credit data */
    DATA credit_raw;
        input account_id $ score $ balance $ review_date $;
        datalines;
    A001 720 5000.00 01JAN2024
    A002 680 3500.50 15JAN2024
    A003 800 7500.75 31JAN2024
    ;
    run;

    /* Process using various conversion functions */
    DATA credit_processed;
        SET credit_raw;

        /* Convert score to numeric */
        score_num = INPUTN(score, 'BEST12.');

        /* Convert balance to numeric with format */
        balance_num = INPUTN(balance, 'COMMA12.2');

        /* Convert date string to SAS date */
        review_date_sas = INPUT(review_date, DATE9.);

        /* Create formatted strings for reporting */
        score_cat = PUT(score_num,
            CASE.
                WHEN (score_num >= 750) 'Excellent'
                WHEN (score_num >= 700) 'Good'
                WHEN (score_num >= 650) 'Fair'
                OTHERWISE 'Poor'
            );

        /* Format balance for display */
        balance_fmt = PUT(balance_num, DOLLAR12.2);

        /* Format date for reporting */
        review_date_fmt = PUT(review_date_sas, DATE9.);
    run;

    /* Create macro variables for reporting */
    PROC SQL NOPRINT;
        SELECT
            PUT(AVG(score_num), 5.1),
            PUT(SUM(balance_num), DOLLAR12.2)
        INTO
```

```
            :avg_score,
            :total_balance
        FROM credit_processed;
    QUIT;

    %put Average Score: &avg_score;
    %put Total Balance: &total_balance;
%mend;

%process_credit_data;
```

6. Error Handling Examples

```
%macro safe_conversions;
    /* Safe numeric conversion */
    %let raw_value = abc123;

    /* Using INPUT with error checking */
    %let numeric_result = %sysfunc(INPUTN(&raw_value, BEST12.));
    %if %sysfunc(missing(&numeric_result)) %then %do;
        %put ERROR: Invalid numeric conversion for &raw_value;
    %end;

    /* Safe date conversion */
    %let date_str = 32JAN2024;

    /* Using INPUTN with date format */
    %let date_result = %sysfunc(INPUTN(&date_str, DATE9.));
    %if %sysfunc(missing(&date_result)) %then %do;
        %put ERROR: Invalid date conversion for &date_str;
    %end;
%mend;

%safe_conversions;
```

7. Common Use Cases Reference Table

Function	Context	Use Case	Example
PUT	Data Step	Format numeric for display	PUT(num, DOLLAR12.2)
PUT	Macro	Convert numeric to character	%sysfunc(PUT(&num, 12.2))

Function	Context	Use Case	Example
PUTN	Data Step	Format numeric with specific format	PUTN(num, 'COMMA12.2')
PUTN	Macro	Format numeric in macro	%sysfunc(PUTN(&num, DOLLAR12.2))
INPUT	Data Step	Convert character to numeric/date	INPUT(char, DATE9.)
INPUT	Macro	Parse string in macro	%sysfunc(INPUT(&str, 12.2))
INPUTN	Data Step	Convert string to numeric	INPUTN(char, 'BEST12.')
INPUTN	Macro	Convert string to numeric in macro	%sysfunc(INPUTN(&str, BEST12.))

Best Practices:

1. Always check for missing values after conversion
2. Use appropriate format specifications
3. Handle errors gracefully in macro context
4. Document format specifications
5. Consider performance implications in large datasets

PROC SQL

1. Basic SQL Operations

```
/* Basic SELECT */
PROC SQL;
    /* Simple select */
    SELECT customer_id, name, credit_score
    FROM customer_data
    WHERE credit_score > 700
    ORDER BY credit_score DESC;

    /* Select with calculations */
```

```
    SELECT
        customer_id,
        balance,
        credit_limit,
        (balance / credit_limit) * 100 as utilization_pct
    FROM credit_accounts
    WHERE calculated utilization_pct > 80;
QUIT;
```

2. Joins

```
/* Different types of joins */
PROC SQL;
    /* Inner Join */
    CREATE TABLE matched_customers AS
    SELECT
        a.customer_id,
        a.name,
        a.credit_score,
        b.balance,
        b.status
    FROM customer_base a
    INNER JOIN account_data b
    ON a.customer_id = b.customer_id;

    /* Left Join */
    CREATE TABLE all_customers AS
    SELECT
        a.customer_id,
        a.name,
        COALESCE(b.balance, 0) as balance
    FROM customer_base a
    LEFT JOIN account_data b
    ON a.customer_id = b.customer_id;

    /* Multiple Joins */
    CREATE TABLE full_profile AS
    SELECT
        a.customer_id,
        a.name,
        b.balance,
        c.credit_score,
        d.payment_history
    FROM customer_base a
    LEFT JOIN account_data b ON a.customer_id = b.customer_id
    LEFT JOIN credit_data c ON a.customer_id = c.customer_id
    LEFT JOIN payment_data d ON a.customer_id = d.customer_id;
QUIT;
```

3. Aggregations and Group By

```
PROC SQL;
    /* Basic aggregation */
    SELECT
        region,
        COUNT(*) as customer_count,
        AVG(credit_score) as avg_score,
        SUM(balance) as total_balance,
        MAX(credit_limit) as max_limit
    FROM customer_data
    GROUP BY region
    HAVING customer_count > 100
    ORDER BY total_balance DESC;

    /* Complex aggregations */
    CREATE TABLE risk_summary AS
    SELECT
        region,
        credit_rating,
        COUNT(DISTINCT customer_id) as unique_customers,
        SUM(balance) as total_exposure,
        AVG(credit_score) as avg_score,
        CALCULATED total_exposure / CALCULATED unique_customers as
avg_exposure,
        SUM(CASE
            WHEN status = 'DEFAULT' THEN balance
            ELSE 0
        END) as default_exposure
    FROM credit_portfolio
    GROUP BY
        region,
        credit_rating
    HAVING unique_customers >= 10;
QUIT;
```

4. Subqueries and Complex Logic

```
PROC SQL;
    /* Subquery in WHERE clause */
    SELECT customer_id, balance
    FROM account_data
    WHERE balance > (
        SELECT AVG(balance) * 2
        FROM account_data
    );
```

```
    /* Correlated subquery */
    SELECT
        a.customer_id,
        a.transaction_date,
        a.amount,
        (SELECT COUNT(*)
         FROM transactions b
         WHERE b.customer_id = a.customer_id
         AND b.transaction_date <= a.transaction_date) as
transaction_sequence
    FROM transactions a;

    /* Multiple subqueries */
    CREATE TABLE high_risk_customers AS
    SELECT
        customer_id,
        name,
        credit_score,
        balance,
        (SELECT AVG(credit_score)
         FROM customer_data) as avg_portfolio_score,
        (SELECT COUNT(*)
         FROM defaults d
         WHERE d.customer_id = a.customer_id) as default_count
    FROM customer_data a
    WHERE credit_score < (
        SELECT
            AVG(credit_score) - STD(credit_score)
        FROM customer_data
    );
QUIT;
```

5. Data Modification

```
PROC SQL;
    /* Insert data */
    INSERT INTO customer_data
    SET customer_id = 'C001',
        name = 'John Doe',
        credit_score = 720;

    /* Insert from query */
    INSERT INTO high_risk_customers
    SELECT *
    FROM customer_data
    WHERE credit_score < 650;

    /* Update records */
```

```
    UPDATE account_data
    SET status = 'INACTIVE'
    WHERE last_transaction_date < '01JAN2024'd;

    /* Delete records */
    DELETE FROM customer_data
    WHERE status = 'CLOSED'
    AND balance = 0;
QUIT;
```

6. Advanced Features

```
PROC SQL;
    /* Using CASE statements */
    SELECT
        customer_id,
        balance,
        CASE
            WHEN credit_score >= 750 THEN 'Excellent'
            WHEN credit_score >= 700 THEN 'Good'
            WHEN credit_score >= 650 THEN 'Fair'
            ELSE 'Poor'
        END as credit_rating,
        CASE
            WHEN balance > credit_limit THEN 'Over Limit'
            WHEN balance/credit_limit > 0.8 THEN 'Near Limit'
            ELSE 'Normal'
        END as utilization_status
    FROM credit_data;

    /* Using SQL functions */
    SELECT
        customer_id,
        UPCASE(name) as customer_name,
        ROUND(balance, 0.01) as rounded_balance,
        DATE() as report_date,
        YEAR(transaction_date) as trans_year,
        MONTH(transaction_date) as trans_month
    FROM customer_data;

    /* Creating macro variables */
    SELECT
        MAX(credit_score) INTO :max_score
    FROM customer_data;

    /* Using macro variables */
    SELECT *
    FROM customer_data
```

```
    WHERE credit_score > &max_score * 0.9;
QUIT;
```

7. Complex Example with Multiple Features

```
/* Comprehensive credit risk analysis */
PROC SQL;
    /* Create temporary table with derived metrics */
    CREATE TABLE work.risk_metrics AS
    SELECT
        a.customer_id,
        a.name,
        a.credit_score,
        b.balance,
        b.credit_limit,
        c.payment_history,

        /* Calculate derived metrics */
        (b.balance / NULLIF(b.credit_limit, 0)) * 100 as
utilization_pct,

        /* Subquery for payment statistics */
        (SELECT COUNT(*)
         FROM payment_history p
         WHERE p.customer_id = a.customer_id
         AND p.status = 'LATE') as late_payments,

        /* Complex CASE logic */
        CASE
            WHEN a.credit_score >= 750 AND
                 calculated utilization_pct < 30 THEN 'Low Risk'
            WHEN a.credit_score >= 700 OR
                 calculated utilization_pct < 50 THEN 'Medium Risk'
            ELSE 'High Risk'
        END as risk_category

    FROM customer_base a
    LEFT JOIN account_data b ON a.customer_id = b.customer_id
    LEFT JOIN payment_data c ON a.customer_id = c.customer_id
    WHERE a.status = 'ACTIVE';

    /* Create summary statistics */
    CREATE TABLE work.risk_summary AS
    SELECT
        risk_category,
        COUNT(*) as customer_count,
        AVG(credit_score) as avg_score,
        SUM(balance) as total_exposure,
```

```
        AVG(utilization_pct) as avg_utilization,
        SUM(late_payments) as total_late_payments
    FROM work.risk_metrics
    GROUP BY risk_category;

    /* Output results to macro variables */
    SELECT
        COUNT(DISTINCT customer_id) INTO :total_customers
    FROM work.risk_metrics;

    SELECT
        SUM(balance) FORMAT=DOLLAR12.2 INTO :total_portfolio
    FROM work.risk_metrics;

    /* Print summary */
    %PUT NOTE: Analyzed &total_customers customers;
    %PUT NOTE: Total portfolio value: &total_portfolio;
QUIT;
```

Best Practices:

1. **Use Appropriate Joins**

```
/* Prefer explicit join syntax */
FROM table1 a
JOIN table2 b ON a.id = b.id
/* Instead of implicit joins */
FROM table1, table2
WHERE table1.id = table2.id
```

2. **Handle Missing Values**

```
/* Use COALESCE or CASE for NULL handling */
SELECT
    customer_id,
    COALESCE(balance, 0) as balance,
    CASE
        WHEN credit_score IS NULL THEN 'Unknown'
        ELSE PUT(credit_score, 3.)
    END as score_category
FROM customer_data;
```

3. **Use Indexes for Performance**

```
/* Create index for frequently joined columns */
CREATE INDEX customer_idx ON customer_data(customer_id);
```

Merging datasets on both axes including handling mismatching column types, and mismatching column lengths

1. Basic Merge Types

A. Horizontal Merge (MERGE Statement)

```
/* Basic merge by ID */
DATA merged_data;
    MERGE dataset1(IN=a)
          dataset2(IN=b);
    BY customer_id;

    /* Keep only matched records */
    IF a AND b;
run;
```

B. Vertical Merge (SET Statement)

```
/* Stacking datasets */
DATA stacked_data;
    SET dataset1
        dataset2;
run;
```

2. Handling Column Type Mismatches

```
/* Create sample datasets with mismatched types */
DATA customer_main;
    INPUT customer_id $ name $ balance;
    DATALINES;
A001 John 1000
A002 Mary 2000
A003 Bob 3000
;
run;

DATA customer_trans;
    INPUT customer_id balance_update;
    DATALINES;
001 1500
002 2500
003 3500
;
run;

/* Solution 1: Convert before merge */
DATA customer_trans_fixed;
    SET customer_trans;
    /* Convert numeric ID to character */
    customer_id = PUT(customer_id, Z3.);
run;

/* Solution 2: Convert during merge */
DATA merged_fix;
    MERGE customer_main(IN=a)
          customer_trans(IN=b
            RENAME=(customer_id=temp_id));
    BY customer_id;

    /* Convert and compare IDs */
    IF b THEN customer_id = PUT(temp_id, Z3.);
run;
```

3. Handling Length Mismatches

```
/* Create datasets with different lengths */
DATA customers1;
    LENGTH customer_id $5 name $10;
    INPUT customer_id $ name $;
    DATALINES;
A0001 John
```

```
A0002 Mary
;
run;

DATA customers2;
    LENGTH customer_id $10 name $20;
    INPUT customer_id $ name $;
    DATALINES;
A0001 John_Smith
A0002 Mary_Jones
;
run;

/* Solution 1: Specify lengths before merge */
DATA merged_length_fix;
    LENGTH customer_id $10 name $20;  /* Use maximum lengths */
    MERGE customers1(IN=a)
          customers2(IN=b);
    BY customer_id;
run;

/* Solution 2: Using PROC CONTENTS to check lengths */
PROC CONTENTS DATA=customers1;
run;
PROC CONTENTS DATA=customers2;
run;
```

4. Complex Merge Scenarios

```
/* Comprehensive merge with multiple issues */
%MACRO merge_with_validation(
    ds1=,
    ds2=,
    key=,
    output=
);
    /* Step 1: Analyze input datasets */
    PROC CONTENTS DATA=&ds1 OUT=contents1 NOPRINT;
    run;
    PROC CONTENTS DATA=&ds2 OUT=contents2 NOPRINT;
    run;

    /* Step 2: Get maximum lengths */
    PROC SQL NOPRINT;
        SELECT MAX(length) INTO :max_length
        FROM (
            SELECT length FROM contents1 WHERE name="&key"
            UNION ALL
```

```
            SELECT length FROM contents2 WHERE name="&key"
        );
    QUIT;

    /* Step 3: Create standardized datasets */
    DATA temp1;
        LENGTH &key $&max_length;
        SET &ds1;
    run;

    DATA temp2;
        LENGTH &key $&max_length;
        SET &ds2;
    run;

    /* Step 4: Perform merge with validation */
    DATA &output;
        MERGE temp1(IN=a)
              temp2(IN=b);
        BY &key;

        /* Track merge status */
        match_status = CASE
            WHEN a AND b THEN 'Matched'
            WHEN a THEN 'Left_Only'
            WHEN b THEN 'Right_Only'
            ELSE 'Unknown'
        END;

        /* Output statistics */
        IF _N_ = 1 THEN DO;
            CALL SYMPUTX('merge_start', PUT(DATETIME(), DATETIME.));
        END;
    run;

    /* Step 5: Report merge results */
    PROC SQL;
        CREATE TABLE merge_stats AS
        SELECT match_status,
               COUNT(*) as record_count
        FROM &output
        GROUP BY match_status;
    QUIT;
%MEND;
```

5. Handling Multiple Types of Mismatches

```
/* Complex merge with multiple data quality issues */
DATA clean_and_merge;
    /* Define maximum lengths for all variables */
    LENGTH
        customer_id $10
        name $50
        status $10
        balance 8
        credit_score 8
    ;

    /* Merge with extensive cleaning */
    MERGE
        customer_base(
            IN=a
            RENAME=(cust_id=customer_id
                    cust_name=name)
        )
        customer_credit(
            IN=b
            RENAME=(id=customer_id
                    score=credit_score)
        );
    BY customer_id;

    /* Handle type conversions */
    IF b AND NOT MISSING(credit_score) THEN DO;
        /* Convert string score to numeric */
        IF NOT INPUT(credit_score, BEST12.) THEN
            credit_score = .;
    END;

    /* Standardize text fields */
    name = PROPCASE(STRIP(name));
    status = UPCASE(STRIP(status));

    /* Validate numeric fields */
    IF NOT MISSING(balance) AND balance < 0 THEN
        balance = 0;

    /* Track merge quality */
    match_type = CASE
        WHEN a AND b THEN 'Full_Match'
        WHEN a THEN 'Base_Only'
        WHEN b THEN 'Credit_Only'
        ELSE 'Unknown'
    END;
```

```
    /* Add audit fields */
    process_date = TODAY();
    process_time = TIME();

    FORMAT
        process_date DATE9.
        process_time TIME8.
        balance DOLLAR12.2
    ;
run;
```

6. Best Practices and Tips

1. **Always Check Data Before Merging**

```
/* Check for duplicates in merge keys */
PROC SQL;
    SELECT customer_id, COUNT(*) as count
    FROM input_data
    GROUP BY customer_id
    HAVING count > 1;
QUIT;
```

2. **Document Length Specifications**

```
/* Clear documentation of length decisions */
LENGTH
    /* Primary keys */
    customer_id $10    /* Max observed length plus buffer */

    /* Name fields */
    first_name $25    /* Standard name length */
    last_name $35     /* Longer for compound names */

    /* Numeric fields */
    balance 8         /* Double precision for calculations */
;
```

3. **Handle Missing Values**

```
/* Proper missing value handling */
IF NOT MISSING(value1) AND NOT MISSING(value2) THEN
    result = value1 + value2;
ELSE
    result = .;
```

4. **Use Appropriate Merge Type**

```
/* Choose appropriate merge type */
DATA merged_data;
    MERGE dataset1(IN=a)
          dataset2(IN=b);
    BY customer_id;

    /* Inner join */
    IF a AND b;

    /* Left join */
    IF a;

    /* Right join */
    IF b;

    /* Full outer join */
    /* Keep all records */
run;
```

SAS string or character functions

1. Basic String Functions

```
DATA string_basics;
    /* Sample strings */
    text = "John Smith";
    email = "john.smith@email.com";
    account = "ACC-123-456";

    /* Basic string functions */
    length_val = LENGTH(text);          /* String length */
    upper_case = UPCASE(text);          /* Convert to uppercase */
    lower_case = LOWCASE(text);         /* Convert to lowercase */
    proper_case = PROPCASE(text);       /* Proper case (First Letter
Cap) */
```

```
    /* Trim functions */
    trimmed = TRIM(text);              /* Remove trailing spaces */
    left_aligned = LEFT(text);         /* Remove leading spaces */
    compressed = COMPRESS(text);        /* Remove all spaces */

    PUT _ALL_;
run;
```

2. String Extraction and Searching

```
DATA string_extract;
    /* Sample text */
    full_name = "John A. Smith";
    email = "john.smith@email.com";

    /* Substring extraction */
    first_char = SUBSTR(full_name, 1, 1);       /* First character */
    first_name = SCAN(full_name, 1, ' ');       /* First word */
    last_name = SCAN(full_name, -1, ' ');       /* Last word */

    /* Finding positions */
    space_pos = FIND(full_name, ' ');           /* Position of first
space */
    at_sign_pos = FIND(email, '@');             /* Position of @ */

    /* Extract domain */
    domain = SUBSTR(email, FIND(email, '@')+1);

    PUT _ALL_;
run;
```

3. Advanced String Manipulation

```
DATA string_advanced;
    /* Sample strings */
    text = "   Credit-Score: 750  ";
    account_num = "ACC123DEF456";

    /* Complex manipulations */
    /* Remove specific characters */
    clean_text = COMPRESS(text, '- :');

    /* Keep only numbers */
    numbers_only = COMPRESS(text, , 'd');
```

```
    /* Keep only letters */
    letters_only = COMPRESS(account_num, , 'a');

    /* Replace characters */
    replaced = TRANWRD(text, 'Credit', 'Risk');

    /* Concatenation */
    concat1 = CATS(text, account_num);    /* Remove all spaces */
    concat2 = CATX(' ', text, account_num); /* Add delimiter */

    PUT _ALL_;
run;
```

4. Pattern Matching and Regular Expressions

```
DATA pattern_matching;
    /* Sample data */
    email = "john.smith@email.com";
    phone = "123-456-7890";

    /* Pattern matching */
    is_email = PRXMATCH('/\w+@\w+\.\w+/', email);
    is_phone = PRXMATCH('/\d{3}-\d{3}-\d{4}/', phone);

    /* Extract using regex */
    if PRXMATCH('/(\w+)@(\w+)\.(\w+)/', email) then do;
        username = PRXPOSN(1, 1, email);
        domain = PRXPOSN(2, 1, email);
        tld = PRXPOSN(3, 1, email);
    end;

    PUT _ALL_;
run;
```

5. String Functions for Credit Risk Analysis

```
DATA credit_string_processing;
    /* Sample credit data */
    customer_id = "  CUS-123-456  ";
    status_code = "ACTIVE/GOOD";
    credit_note = "Score:720|Limit:5000|Status:OK";

    /* Clean customer ID */
    clean_id = COMPRESS(UPCASE(customer_id), " -");

    /* Split status code */
```

```
    account_status = SCAN(status_code, 1, '/');
    credit_status = SCAN(status_code, 2, '/');

    /* Parse credit note */
    score = INPUT(SCAN(SCAN(credit_note, 1, '|'), 2, ':'), 8.);
    limit = INPUT(SCAN(SCAN(credit_note, 2, '|'), 2, ':'), 8.);
    status = SCAN(SCAN(credit_note, 3, '|'), 2, ':');

    PUT _ALL_;
run;
```

6. Common String Function Reference Table

Function	Purpose	Example	Result
LENGTH	Get string length	LENGTH("ABC")	3
UPCASE	Convert to uppercase	UPCASE("abc")	"ABC"
LOWCASE	Convert to lowercase	LOWCASE("ABC")	"abc"
PROPCASE	Convert to proper case	PROPCASE("john smith")	"John Smith"
TRIM	Remove trailing spaces	TRIM(" ABC ")	" ABC"
LEFT	Remove leading spaces	LEFT(" ABC")	"ABC "
COMPRESS	Remove characters	COMPRESS("A B C")	"ABC"
SUBSTR	Extract substring	SUBSTR("ABCDE", 2, 3)	"BCD"
SCAN	Extract word	SCAN("A B C", 2)	"B"
FIND	Find position	FIND("ABC", "B")	2
TRANWRD	Replace text	TRANWRD("ABC", "B", "X")	"AXC"
CATS	Concatenate, remove spaces	CATS("A ", " B")	"AB"
CATX	Concatenate with delimiter	CATX(",", "A", "B")	"A,B"

7. Practical Examples

```
/* Complex string processing for credit data */
DATA credit_processing;
    /* Input data */
    input @1 raw_record $100.;
    datalines;
CUS-001|JOHN SMITH|SCORE:720|STATUS:ACTIVE
CUS-002|MARY JONES|SCORE:680|STATUS:REVIEW
CUS-003|BOB WILSON|SCORE:800|STATUS:ACTIVE
;

    /* Process each record */
    /* Extract customer ID */
    customer_id = COMPRESS(SCAN(raw_record, 1, '|'), '-');

    /* Clean and format name */
    full_name = PROPCASE(TRIM(SCAN(raw_record, 2, '|')));

    /* Extract score */
    score_text = SCAN(raw_record, 3, '|');
    credit_score = INPUT(COMPRESS(score_text, 'SCORE:'), 8.);

    /* Extract and clean status */
    status_text = SCAN(raw_record, 4, '|');
    account_status = COMPRESS(status_text, 'STATUS:');

    /* Create formatted output */
    formatted_record = CATX('|',
        PUT(customer_id, $8.),
        PUT(full_name, $20.),
        PUT(credit_score, 4.),
        PUT(account_status, $10.)
    );

    PUT formatted_record=;
run;
```

8. Error Handling with String Functions

```
DATA string_error_handling;
    /* Sample data with potential issues */
    text1 = "ABC";
    text2 = "";
    text3 = " ";

    /* Safe string handling */
```

```
    /* Check for missing or empty strings */
    if MISSING(text1) then
        clean_text1 = "MISSING";
    else
        clean_text1 = TRIM(text1);

    /* Combine multiple checks */
    length_check = IFC(LENGTH(TRIM(text2)) > 0,
                       "Has Content",
                       "Empty");

    /* Safe substring extraction */
    safe_substr = IFC(LENGTH(text3) >= 3,
                      SUBSTR(text3, 1, 3),
                      text3);

    PUT _ALL_;
run;
```

Best Practices:

1. Always handle missing values
2. Use appropriate string functions for the task
3. Consider case sensitivity
4. Use efficient string operations for large datasets
5. Properly allocate string lengths
6. Document complex string manipulations

SAS date and time functions

- **Finding interval difference between two dates at different granularity**
- **Finding the date some period in past or future from a particular date**
- **All other functions**

1. Basic Date and Time Functions

```
DATA date_basics;
    /* Current date and time */
    current_date = TODAY();          /* Current date */
    current_datetime = DATETIME();   /* Current datetime */
```

```
    current_time = TIME();           /* Current time */

    /* Format specifications */
    FORMAT
        current_date DATE9.
        current_datetime DATETIME20.
        current_time TIME8.
    ;

    PUT _ALL_;
run;
```

2. Interval Differences Between Dates

```
DATA date_intervals;
    /* Sample dates */
    start_date = '01JAN2024'd;
    end_date = '31DEC2024'd;

    /* Different interval calculations */
    days_between = end_date - start_date;
    months_between = INTCK('MONTH', start_date, end_date);
    years_between = INTCK('YEAR', start_date, end_date);
    quarters_between = INTCK('QTR', start_date, end_date);
    weeks_between = INTCK('WEEK', start_date, end_date);
    weekdays_between = INTCK('WEEKDAY', start_date, end_date);

    /* Age calculation */
    birth_date = '01JAN1990'd;
    age_years = YRDIF(birth_date, TODAY(), 'AGE');

    FORMAT start_date end_date birth_date DATE9.;
    PUT _ALL_;
run;
```

3. Date Shifting (Past and Future Dates)

```
DATA date_shifting;
    base_date = '15JAN2024'd;

    /* Future dates */
    next_day = INTNX('DAY', base_date, 1);
    next_week = INTNX('WEEK', base_date, 1);
    next_month = INTNX('MONTH', base_date, 1);
    next_quarter = INTNX('QTR', base_date, 1);
    next_year = INTNX('YEAR', base_date, 1);
```

```
    /* Past dates */
    prev_day = INTNX('DAY', base_date, -1);
    prev_week = INTNX('WEEK', base_date, -1);
    prev_month = INTNX('MONTH', base_date, -1);
    prev_quarter = INTNX('QTR', base_date, -1);
    prev_year = INTNX('YEAR', base_date, -1);

    /* Beginning and end of periods */
    month_start = INTNX('MONTH', base_date, 0, 'B');
    month_end = INTNX('MONTH', base_date, 0, 'E');
    quarter_start = INTNX('QTR', base_date, 0, 'B');
    quarter_end = INTNX('QTR', base_date, 0, 'E');

    FORMAT _ALL_ DATE9.;
    PUT _ALL_;
run;
```

4. Date Components and Extraction

```
DATA date_components;
    sample_date = '15JAN2024'd;

    /* Extract components */
    year_num = YEAR(sample_date);
    month_num = MONTH(sample_date);
    day_num = DAY(sample_date);

    /* Day of week */
    weekday_num = WEEKDAY(sample_date);    /* 1=Sunday, 2=Monday, etc.
*/

    /* Week number */
    week_num = WEEK(sample_date);
    week_iso = WEEK(sample_date, 'V');     /* ISO week number */

    /* Quarter */
    quarter_num = QTR(sample_date);

    /* Beginning and end of year */
    year_start = INTNX('YEAR', sample_date, 0, 'B');
    year_end = INTNX('YEAR', sample_date, 0, 'E');

    FORMAT sample_date year_start year_end DATE9.;
    PUT _ALL_;
run;
```

5. Complex Date Calculations for Credit Risk

```
DATA credit_dates;
    /* Account dates */
    account_open_date = '01JAN2023'd;
    last_payment_date = '15JAN2024'd;
    review_date = TODAY();

    /* Calculate various durations */
    account_age_days = review_date - account_open_date;
    account_age_months = INTCK('MONTH', account_open_date, review_date);
    months_since_payment = INTCK('MONTH', last_payment_date,
review_date);

    /* Payment due dates */
    next_payment_date = INTNX('MONTH', last_payment_date, 1);
    grace_period_end = INTNX('DAY', next_payment_date, 15);

    /* Reporting periods */
    report_start = INTNX('MONTH', review_date, -12, 'B');  /* Last 12
months */
    report_end = INTNX('MONTH', review_date, 0, 'E');      /* Current
month end */

    /* Aging buckets */
    SELECT;
        WHEN (months_since_payment = 0) aging_bucket = 'Current';
        WHEN (months_since_payment = 1) aging_bucket = '30 Days';
        WHEN (months_since_payment = 2) aging_bucket = '60 Days';
        WHEN (months_since_payment = 3) aging_bucket = '90 Days';
        OTHERWISE aging_bucket = '90+ Days';
    END;

    FORMAT account_open_date last_payment_date review_date
           next_payment_date grace_period_end
           report_start report_end DATE9.;

    PUT _ALL_;
run;
```

6. Working with Time Components

```
DATA time_calculations;
    current_time = TIME();
    current_datetime = DATETIME();

    /* Extract time components */
```

```
    hour_val = HOUR(current_time);
    minute_val = MINUTE(current_time);
    second_val = SECOND(current_time);

    /* Time intervals */
    future_time = current_time + (60*60);  /* Add 1 hour */
    time_diff = future_time - current_time;

    /* DateTime calculations */
    future_datetime = INTNX('HOUR', current_datetime, 24);  /* Add 24
hours */
    datetime_diff = future_datetime - current_datetime;

    FORMAT
        current_time future_time TIME8.
        current_datetime future_datetime DATETIME20.
    ;

    PUT _ALL_;
run;
```

7. Date/Time Function Reference Table

Function	Purpose	Example	Result
TODAY()	Current date	TODAY()	Current date
DATETIME()	Current datetime	DATETIME()	Current datetime
TIME()	Current time	TIME()	Current time
INTCK	Count intervals	INTCK('MONTH', date1, date2)	Number of months
INTNX	Shift date by interval	INTNX('MONTH', date, 1)	Next month
YRDIF	Calculate years between	YRDIF(birth_date, today, 'AGE')	Age in years
YEAR	Extract year	YEAR(date)	Year number
MONTH	Extract month	MONTH(date)	Month number
DAY	Extract day	DAY(date)	Day number

Function	Purpose	Example	Result
WEEKDAY	Get day of week	WEEKDAY(date)	1-7 (Sun-Sat)
QTR	Get quarter	QTR(date)	1-4
WEEK	Get week number	WEEK(date)	Week number

8. Common Date Formats

```
DATA date_formats;
    sample_date = '15JAN2024'd;

    /* Different date formats */
    PUT sample_date DATE9.;         /* 15JAN2024 */
    PUT sample_date DDMMYY10.;      /* 15/01/2024 */
    PUT sample_date MMDDYY10.;      /* 01/15/2024 */
    PUT sample_date YYMMDD10.;      /* 2024-01-15 */
    PUT sample_date WEEKDATE.;      /* Monday, January 15, 2024 */
    PUT sample_date WORDDATX.;      /* January 15, 2024 */
    PUT sample_date MONYY7.;        /* JAN2024 */
run;
```

Best Practices:

1. Always use appropriate formats for date display
2. Consider timezone implications
3. Handle missing dates appropriately
4. Use consistent date formats across analysis
5. Document date calculations
6. Consider business days vs calendar days
7. Handle leap years appropriately

Sorting a dataset and De-duplicating a dataset on different set of columns or entire dataset

1. Basic Sorting Using PROC SORT

```
/* Basic sorting */
PROC SORT DATA=input_data OUT=sorted_data;
    BY customer_id transaction_date;
run;

/* Sorting in descending order */
PROC SORT DATA=input_data OUT=sorted_data;
    BY DESCENDING balance customer_id;
run;

/* Sorting with multiple orders */
PROC SORT DATA=input_data OUT=sorted_data;
    BY DESCENDING credit_score
       customer_id
       DESCENDING transaction_date;
run;
```

2. De-duplication Using NODUPKEY/NODUPREC

```
/* Remove exact duplicates (all columns) */
PROC SORT DATA=input_data OUT=unique_data NODUPREC;
    BY _ALL_;
run;

/* Remove duplicates based on specific keys */
PROC SORT DATA=input_data OUT=unique_data NODUPKEY;
    BY customer_id transaction_date;
run;
```

3. Comprehensive Example with Credit Data

```
/* Create sample credit data */
DATA credit_transactions;
    INPUT customer_id $ transaction_date :DATE9.
          amount status $ branch $;
    FORMAT transaction_date DATE9.;
    DATALINES;
C001 01JAN2024 1000 ACTIVE NY
C001 01JAN2024 1000 ACTIVE NY
C001 15JAN2024 2000 ACTIVE NY
C002 01JAN2024 3000 REVIEW LA
C002 01JAN2024 3000 REVIEW LA
C003 01JAN2024 4000 ACTIVE CH
;
run;

/* 1. Sort and remove exact duplicates */
PROC SORT DATA=credit_transactions
          OUT=unique_transactions NODUPREC;
    BY _ALL_;
run;

/* 2. Keep only latest transaction per customer */
PROC SORT DATA=credit_transactions
          OUT=latest_transactions NODUPKEY;
    BY customer_id DESCENDING transaction_date;
run;

/* 3. Keep first transaction per customer-branch combination */
PROC SORT DATA=credit_transactions
          OUT=first_transactions NODUPKEY;
    BY customer_id branch transaction_date;
run;
```

4. Advanced De-duplication Using DATA Step

```
/* De-duplication with custom logic */
DATA custom_dedup;
    SET credit_transactions;
    BY customer_id transaction_date;

    /* Keep first record of the day with highest amount */
    IF FIRST.transaction_date THEN DO;
        max_amount = amount;
        output;
    END;
```

```
    ELSE DO;
        IF amount > max_amount THEN DO;
            max_amount = amount;
            output;
        END;
    END;
run;
```

5. Using FIRST. and LAST. Variables

```
/* Using FIRST. and LAST. for complex de-duplication */
PROC SORT DATA=credit_transactions;
    BY customer_id transaction_date amount;
run;

DATA first_last_example;
    SET credit_transactions;
    BY customer_id transaction_date;

    /* First transaction of the day per customer */
    IF FIRST.transaction_date THEN
        output;

    /* Last transaction of the day per customer */
    IF LAST.transaction_date THEN
        output;

    /* Both first and last if different */
    IF FIRST.transaction_date OR LAST.transaction_date THEN
        output;
run;
```

6. Using SQL for De-duplication

```
/* Using PROC SQL for de-duplication */
PROC SQL;
    /* Keep one record per customer with maximum amount */
    CREATE TABLE sql_dedup AS
    SELECT *
    FROM credit_transactions
    WHERE (customer_id, amount) IN (
        SELECT customer_id,
               MAX(amount) as max_amount
        FROM credit_transactions
        GROUP BY customer_id
    );
```

```
    /* Alternative approach using partition */
    CREATE TABLE sql_dedup2 AS
    SELECT *
    FROM (
        SELECT *,
               ROW_NUMBER() OVER (
                   PARTITION BY customer_id
                   ORDER BY transaction_date DESC
               ) as row_num
        FROM credit_transactions
    )
    WHERE row_num = 1;
QUIT;
```

7. Complex Sorting and De-duplication

```
/* Complex example combining multiple techniques */
/* First, create detailed transaction data */
DATA detailed_transactions;
    INPUT customer_id $
          transaction_date :DATE9.
          amount
          status $
          risk_score
          branch $;
    FORMAT transaction_date DATE9.;
    DATALINES;
C001 01JAN2024 1000 ACTIVE 720 NY
C001 01JAN2024 1500 ACTIVE 720 NY
C001 15JAN2024 2000 REVIEW 700 NY
C002 01JAN2024 3000 ACTIVE 680 LA
C002 01JAN2024 3500 REVIEW 670 LA
C002 15JAN2024 4000 ACTIVE 690 LA
;
run;

/* Multi-step de-duplication process */
/* Step 1: Sort by relevant fields */
PROC SORT DATA=detailed_transactions;
    BY customer_id
       DESCENDING risk_score
       DESCENDING transaction_date;
run;

/* Step 2: Complex de-duplication logic */
DATA final_dedup;
    SET detailed_transactions;
```

```
    BY customer_id;

    /* Initialize variables for first customer */
    IF FIRST.customer_id THEN DO;
        total_amount = 0;
        transaction_count = 0;
    END;

    /* Accumulate totals */
    total_amount + amount;
    transaction_count + 1;

    /* Output logic */
    IF LAST.customer_id THEN DO;
        avg_amount = total_amount / transaction_count;
        OUTPUT;
    END;

    /* Keep track of running totals */
    RETAIN total_amount transaction_count;
run;
```

8. Best Practices and Tips

1. **Always Check Results**

```
/* Check record counts before and after */
PROC SQL;
    SELECT COUNT(*) as total_records,
           COUNT(DISTINCT customer_id) as unique_customers
    FROM input_data;
QUIT;
```

2. **Document Sort Orders**

```
/* Clear documentation of sort criteria */
PROC SORT DATA=input_data OUT=sorted_data;
    BY /* Primary key */ customer_id
       /* Time dimension */ DESCENDING transaction_date
       /* Business priority */ DESCENDING amount;
run;
```

3. **Handle Missing Values**

```
/* Sort with missing values consideration */
PROC SORT DATA=input_data OUT=sorted_data MISSING;
    BY customer_id transaction_date;
run;
```

4. **Performance Optimization**

```
/* Use SORTSIZE= for large datasets */
OPTIONS SORTSIZE=MAX;
PROC SORT DATA=large_dataset OUT=sorted_data;
    BY customer_id;
run;
```

Common Pitfalls to Avoid:

1. Not checking for missing values
2. Forgetting to verify record counts
3. Not considering sort order impact on subsequent processing
4. Using unnecessary sorting when SQL might be more efficient
5. Not documenting de-duplication logic

Macro Programming

- **Macro Variables (global or local). Creation, Assignment, Dereference, Print**
- **Macro functions**

1. Macro Variables - Basic Creation and Usage

```
/* Global Macro Variables */
%LET global_var = Global Value;    /* Global scope */

/* Within a macro (Local Macro Variables) */
%MACRO example_macro;
    %LOCAL local_var;              /* Local scope */
    %LET local_var = Local Value;
run;

/* Different ways to create macro variables */
/* 1. Using %LET */
%LET customer_id = C001;
```

```
/* 2. Using CALL SYMPUT in DATA step */
DATA _NULL_;
    x = 100;
    CALL SYMPUT('data_var', PUT(x, 8.));
run;

/* 3. Using SQL */
PROC SQL;
    SELECT MAX(balance) INTO :max_balance
    FROM account_data;
QUIT;

/* 4. Using SYMPUTX (handles numeric better) */
DATA _NULL_;
    score = 750;
    CALL SYMPUTX('credit_score', score);
run;
```

2. Referencing Macro Variables

```
/* Basic referencing */
%PUT &global_var;          /* Direct reference */
%PUT &&dynamic&i;         /* Double ampersand for dynamic reference */

/* Complex referencing */
%LET prefix = customer;
%LET suffix = _id;
%PUT &prefix&suffix;      /* Concatenated reference */

/* Using macro variables in code */
DATA filtered_data;
    SET full_data;
    WHERE credit_score > &min_score;
run;
```

3. Macro Functions

```
/* String manipulation */
%LET text = Credit Score;
%LET upper_text = %UPCASE(&text);
%LET lower_text = %LOWCASE(&text);

/* Numeric functions */
%LET num = 123.456;
%LET rounded = %SYSFUNC(ROUND(&num, 0.01));
```

```
/* Date functions */
%LET today = %SYSFUNC(TODAY(), DATE9.);
%LET next_month = %SYSFUNC(INTNX(MONTH, %SYSFUNC(TODAY()), 1), DATE9.);

/* Evaluation functions */
%LET condition = %EVAL(10 > 5);
%LET complex_calc = %SYSEVALF(10.5 * 2.3);

/* String functions within macros */
%MACRO string_ops;
    %LET length = %LENGTH(&text);
    %LET substr = %SUBSTR(&text, 1, 5);
    %PUT Length: &length Substring: &substr;
%MEND;
```

4. Complex Macro Example for Credit Risk Analysis

```
/* Comprehensive credit risk macro */
%MACRO analyze_credit_risk(
    input_ds=,           /* Input dataset */
    score_threshold=,    /* Credit score threshold */
    date_from=,          /* Analysis start date */
    date_to=,            /* Analysis end date */
    output_ds=           /* Output dataset */
);

    /* Local variable declaration */
    %LOCAL record_count avg_score;

    /* Get basic statistics */
    PROC SQL NOPRINT;
        SELECT COUNT(*) FORMAT=COMMA12.,
               AVG(credit_score) FORMAT=8.2
        INTO :record_count, :avg_score
        FROM &input_ds
        WHERE credit_score >= &score_threshold AND
              review_date BETWEEN "&date_from"d AND "&date_to"d;
    QUIT;

    /* Log information */
    %PUT NOTE: Processing &record_count records;
    %PUT NOTE: Average credit score: &avg_score;

    /* Main processing */
    DATA &output_ds;
```

```
        SET &input_ds;
        WHERE credit_score >= &score_threshold AND
              review_date BETWEEN "&date_from"d AND "&date_to"d;

        /* Risk categorization */
        %IF &avg_score > 700 %THEN %DO;
            risk_factor = 'LOW';
        %END;
        %ELSE %DO;
            risk_factor = 'HIGH';
        %END;
    run;

    /* Return status */
    %IF &SQLOBS > 0 %THEN %DO;
        %PUT NOTE: Processing completed successfully;
    %END;
    %ELSE %DO;
        %PUT ERROR: No records processed;
    %END;

%MEND analyze_credit_risk;

/* Usage example */
%analyze_credit_risk(
    input_ds=credit_data,
    score_threshold=650,
    date_from=01JAN2024,
    date_to=31JAN2024,
    output_ds=risk_analysis
);
```

5. Advanced Macro Techniques

```
/* 1. Dynamic variable creation */
%MACRO create_vars;
    %DO i = 1 %TO 5;
        var&i = &i;
    %END;
%MEND;

/* 2. Macro loops */
%MACRO process_months;
    %DO month = 1 %TO 12;
        %LET date = %SYSFUNC(MDY(&month, 1, 2024), DATE9.);
        %PUT Processing month: &date;
    %END;
%MEND;
```

```
/* 3. Conditional processing */
%MACRO check_data(value=);
    %IF %SYSEVALF(&value > 100) %THEN %DO;
        %PUT Value exceeds threshold;
    %END;
    %ELSE %DO;
        %PUT Value within limits;
    %END;
%MEND;

/* 4. List processing */
%MACRO process_list(list=);
    %LET count = %SYSFUNC(COUNTW(&list));
    %DO i = 1 %TO &count;
        %LET item = %SCAN(&list, &i);
        %PUT Processing item: &item;
    %END;
%MEND;
```

6. Error Handling and Debugging

```
%MACRO safe_processing(param=);
    /* Parameter validation */
    %IF %LENGTH(&param) = 0 %THEN %DO;
        %PUT ERROR: Parameter is required;
        %RETURN;
    %END;

    /* Try-catch style error handling */
    %LET error_flag = 0;

    DATA _NULL_;
        IF &error_flag = 0 THEN DO;
            /* Processing logic */
        END;
        ELSE DO;
            PUT "ERROR: Processing failed";
        END;
    run;

    /* Debug information */
    %PUT DEBUG: Macro variables at exit:;
    %PUT _LOCAL_;    /* Print local macro variables */
    %PUT _GLOBAL_;   /* Print global macro variables */
%MEND;
```

7. Utility Macros

```
/* Check dataset existence */
%MACRO dataset_exists(dsn);
    %LOCAL rc;
    %LET rc = %SYSFUNC(EXIST(&dsn));
    &rc
%MEND;

/* Format number with commas */
%MACRO format_number(num);
    %SYSFUNC(PUTN(&num, COMMA12.))
%MEND;

/* Date validation */
%MACRO is_valid_date(date_str);
    %SYSFUNC(INPUT(&date_str, YYMMDD10.))
%MEND;

/* String manipulation utilities */
%MACRO clean_string(str);
    %SYSFUNC(COMPRESS(%UPCASE(&str), %STR( )))
%MEND;
```

Best Practices:

1. **Documentation**

```
%MACRO documented_macro(param1=, param2=) / DES='Purpose of macro';
    /* Parameter validation */
    /* Processing logic */
    /* Return values */
%MEND;
```

2. **Parameter Validation**

```
%MACRO validate_params(param=);
    %IF %SUPERQ(param) = %STR() %THEN %DO;
        %PUT ERROR: Parameter is required;
        %RETURN;
    %END;
%MEND;
```

3. **Scope Control**

```
%MACRO scope_example;
    %LOCAL var1 var2;  /* Explicitly declare local variables */
    %GLOBAL result;    /* Explicitly declare global variables */
%MEND;
```

Creating custom macro functions

1. Basic Macro Function Structure

```
/* Basic macro function template */
%MACRO function_name(param1=, param2=) / PARMBUFF;
    /* Returns a value */
    &result
%MEND function_name;
```

2. Simple Custom Functions

```
/* 1. Calculate Age from Date of Birth */
%MACRO calc_age(dob=, as_of_date=);
    %LOCAL age;
    %LET age = %SYSFUNC(FLOOR(
        %SYSFUNC(YRDIF(
            %SYSFUNC(INPUT(&dob, YYMMDD10.)),
            %SYSFUNC(INPUT(&as_of_date, YYMMDD10.)),
            AGE
        ))
    ));
    &age
%MEND calc_age;

/* Usage */
%LET customer_age = %calc_age(
    dob=1990-01-01,
    as_of_date=2024-01-29
);
%PUT Customer Age: &customer_age;

/* 2. Format Number with Commas */
%MACRO format_num(number=);
    %SYSFUNC(STRIP(
        %SYSFUNC(PUTN(&number, COMMA20.2))
    ))
```

```
%MEND format_num;

/* Usage */
%LET formatted = %format_num(number=1234567.89);
%PUT Formatted Number: &formatted;
```

3. String Manipulation Functions

```
/* 1. Clean String (Remove Special Characters) */
%MACRO clean_string(text=);
    %SYSFUNC(COMPRESS(
        %SYSFUNC(STRIP(&text)),
        %STR(,.-_)
    ))
%MEND clean_string;

/* 2. Proper Case Function */
%MACRO proper_case(text=);
    %SYSFUNC(PROPCASE(
        %SYSFUNC(STRIP(&text))
    ))
%MEND proper_case;

/* 3. Extract Word from String */
%MACRO get_word(text=, position=);
    %SYSFUNC(SCAN(&text, &position, %STR( )))
%MEND get_word;

/* Usage Examples */
%LET clean_text = %clean_string(text=John-Doe, Jr.);
%LET proper_name = %proper_case(text=JOHN DOE);
%LET first_name = %get_word(text=John Doe Smith, position=1);
```

4. Date Manipulation Functions

```
/* 1. Add Months to Date */
%MACRO add_months(date=, months=);
    %SYSFUNC(PUTN(
        %SYSFUNC(INTNX(MONTH,
            %SYSFUNC(INPUT(&date, YYMMDD10.)),
            &months
        )),
        YYMMDD10.
    ))
%MEND add_months;
```

```
/* 2. Get End of Month */
%MACRO end_of_month(date=);
    %SYSFUNC(PUTN(
        %SYSFUNC(INTNX(MONTH,
            %SYSFUNC(INPUT(&date, YYMMDD10.)),
            0,
            END
        )),
        YYMMDD10.
    ))
%MEND end_of_month;

/* 3. Date Difference in Months */
%MACRO months_between(date1=, date2=);
    %SYSFUNC(INTCK(MONTH,
        %SYSFUNC(INPUT(&date1, YYMMDD10.)),
        %SYSFUNC(INPUT(&date2, YYMMDD10.))
    ))
%MEND months_between;
```

5. Validation Functions

```
/* 1. Validate Date Format */
%MACRO is_valid_date(date=);
    %LOCAL result;
    %LET result = %SYSFUNC(NOTMISS(
        %SYSFUNC(INPUT(&date, YYMMDD10.))
    ));
    &result
%MEND is_valid_date;

/* 2. Validate Numeric Value */
%MACRO is_numeric(value=);
    %SYSFUNC(VERIFY(&value, 0123456789.))=0
%MEND is_numeric;

/* 3. Validate Email Format */
%MACRO is_valid_email(email=);
    %SYSFUNC(PRXMATCH(
        /^[a-zA-Z0-9._%+-]+@[a-zA-Z0-9.-]+\.[a-zA-Z]{2,}$/,
        &email
    ))
%MEND is_valid_email;
```

6. Complex Business Logic Functions

```
/* 1. Calculate Risk Score */
%MACRO calc_risk_score(
    credit_score=,
    income=,
    debt_ratio=
);
    %LOCAL risk_score;
    %LET risk_score = %SYSEVALF(
        (&credit_score * 0.5) +
        ((&income / 1000) * 0.3) +
        ((100 - &debt_ratio) * 0.2)
    );
    %SYSFUNC(ROUND(&risk_score, 0.01))
%MEND calc_risk_score;

/* 2. Determine Credit Category */
%MACRO credit_category(score=);
    %IF &score >= 750 %THEN %DO;
        Excellent
    %END;
    %ELSE %IF &score >= 700 %THEN %DO;
        Good
    %END;
    %ELSE %IF &score >= 650 %THEN %DO;
        Fair
    %END;
    %ELSE %DO;
        Poor
    %END;
%MEND credit_category;
```

7. Utility Functions

```
/* 1. Dataset Existence Check */
%MACRO dataset_exists(libname=WORK, dsname=);
    %SYSFUNC(EXIST(&libname..&dsname))
%MEND dataset_exists;

/* 2. Variable Existence Check */
%MACRO var_exists(dsname=, varname=);
    %SYSFUNC(VARNUM(&dsname, &varname))>0
%MEND var_exists;

/* 3. Get Dataset Record Count */
%MACRO get_nobs(dsname=);
```

```
    %LOCAL dsid nobs rc;
    %LET dsid = %SYSFUNC(OPEN(&dsname));
    %LET nobs = %SYSFUNC(ATTRN(&dsid, NOBS));
    %LET rc = %SYSFUNC(CLOSE(&dsid));
    &nobs
%MEND get_nobs;
```

8. Error Handling Functions

```
/* 1. Safe Division Function */
%MACRO safe_divide(numerator=, denominator=);
    %IF &denominator = 0 %THEN %DO;
        0
    %END;
    %ELSE %DO;
        %SYSEVALF(&numerator / &denominator)
    %END;
%MEND safe_divide;

/* 2. Try-Catch Style Function */
%MACRO try_function(func=);
    %LOCAL rc result;
    %LET rc = 0;

    %SYSCALL SET(rc);
    %LET result = &func;

    %IF &rc = 0 %THEN %DO;
        &result
    %END;
    %ELSE %DO;
        ERROR
    %END;
%MEND try_function;
```

Best Practices:

1. **Documentation**

```
%MACRO documented_function(param1=, param2=) /
    DES="Function description"
    PARMBUFF;

    /* Parameter validation */
    %IF %SUPERQ(param1) = %STR() %THEN %DO;
        %PUT ERROR: param1 is required;
        %RETURN;
    %END;

    /* Function logic */
    &result
%MEND documented_function;
```

2. **Parameter Validation**

```
%MACRO validated_function(param=);
    /* Type checking */
    %IF %DATATYP(&param) NE NUMERIC %THEN %DO;
        %PUT ERROR: Parameter must be numeric;
        %RETURN;
    %END;

    /* Range checking */
    %IF &param < 0 OR &param > 100 %THEN %DO;
        %PUT ERROR: Parameter must be between 0 and 100;
        %RETURN;
    %END;
%MEND validated_function;
```

Loops within data step and within Macros

1. DATA Step Loops

A. DO Loop (Simple Counter)

```
DATA do_loop;
    /* Basic counter loop */
    DO i = 1 TO 5;
        output;
```

```
    END;
run;

/* With increment */
DATA do_loop_increment;
    DO i = 0 TO 10 BY 2;
        value = i * 2;
        output;
    END;
run;
```

B. DO WHILE Loop

```
DATA while_loop;
    balance = 1000;
    interest_rate = 0.05;
    year = 0;

    /* Continue until balance doubles */
    DO WHILE(balance < 2000);
        year + 1;
        balance = balance * (1 + interest_rate);
        output;
    END;
run;
```

C. DO UNTIL Loop

```
DATA until_loop;
    payment = 100;
    debt = 1000;
    month = 0;

    /* Continue until debt is paid */
    DO UNTIL(debt <= 0);
        month + 1;
        debt = debt - payment;
        output;
    END;
run;
```

D. Nested DO Loops

```
DATA nested_loops;
    /* Create a multiplication table */
    DO i = 1 TO 5;
        DO j = 1 TO 5;
            product = i * j;
            output;
        END;
    END;
run;
```

E. Complex DATA Step Loop Example

```
/* Credit payment simulation */
DATA payment_schedule;
    /* Loan parameters */
    loan_amount = 10000;
    annual_rate = 0.05;
    monthly_rate = annual_rate/12;
    payment = 500;

    /* Initialize variables */
    remaining_balance = loan_amount;
    month = 0;
    total_interest = 0;

    /* Continue until loan is paid off */
    DO UNTIL(remaining_balance <= 0);
        month + 1;

        /* Calculate interest */
        monthly_interest = remaining_balance * monthly_rate;
        total_interest + monthly_interest;

        /* Apply payment */
        principal_payment = MIN(payment, remaining_balance +
monthly_interest);
        remaining_balance = remaining_balance + monthly_interest -
principal_payment;

        /* Output monthly status */
        OUTPUT;
    END;

    FORMAT remaining_balance monthly_interest principal_payment
DOLLAR12.2;
run;
```

2. Macro Loops

A. %DO Loop

```
%MACRO basic_macro_loop;
    %DO i = 1 %TO 5;
        %PUT Iteration &i;
    %END;
%MEND;
%basic_macro_loop;
```

B. %DO %WHILE Loop

```
%MACRO while_macro_loop;
    %LET counter = 1;
    %DO %WHILE(&counter <= 5);
        %PUT Counter: &counter;
        %LET counter = %EVAL(&counter + 1);
    %END;
%MEND;
%while_macro_loop;
```

C. %DO %UNTIL Loop

```
%MACRO until_macro_loop;
    %LET value = 1;
    %DO %UNTIL(&value > 10);
        %PUT Value: &value;
        %LET value = %EVAL(&value * 2);
    %END;
%MEND;
%until_macro_loop;
```

D. Complex Macro Loop Examples

```
/* 1. Create Multiple Datasets */
%MACRO create_monthly_data;
    %DO month = 1 %TO 12;
        /* Format month for dataset name */
        %LET month_fmt = %SYSFUNC(PUT(&month, Z2.));

        DATA monthly_data_&month_fmt;
            SET full_data;
            WHERE MONTH(date) = &month;
```

```
        run;

        %PUT Created dataset for month &month_fmt;
    %END;
%MEND;

/* 2. Process Date Ranges */
%MACRO process_date_range(start_date=, end_date=);
    %LET current_date = &start_date;

    %DO %UNTIL(&current_date > &end_date);
        /* Process data for current date */
        DATA work.temp_&current_date;
            SET main_data;
            WHERE date = "&current_date"d;
        run;

        /* Move to next date */
        %LET current_date = %SYSFUNC(PUTC(
            %SYSFUNC(INTNX(DAY, "&current_date"d, 1)), YYMMDD10.
        ));
    %END;
%MEND;

/* 3. Dynamic Variable Creation */
%MACRO create_variables;
    DATA dynamic_vars;
        %DO i = 1 %TO 12;
            var_&i = 0;
            %PUT Created variable: var_&i;
        %END;
    run;
%MEND;
```

3. Combined DATA Step and Macro Loops

```
/* Complex credit risk analysis example */
%MACRO analyze_risk_segments(
    input_ds=,
    score_ranges=,
    output_prefix=
);
    /* Parse score ranges */
    %LET num_ranges = %SYSFUNC(COUNTW(&score_ranges, |));

    /* Process each score range */
    %DO i = 1 %TO &num_ranges;
        %LET range = %SCAN(&score_ranges, &i, |);
```

```
        %LET min_score = %SCAN(&range, 1, -);
        %LET max_score = %SCAN(&range, 2, -);

        /* Create dataset for this range */
        DATA &output_prefix._&i;
            SET &input_ds;
            WHERE credit_score BETWEEN &min_score AND &max_score;

            /* Calculate risk metrics */
            DO month = 1 TO 12;
                projected_risk = credit_score * (1 - (month * 0.01));
                month_date = INTNX('MONTH', TODAY(), month);
                OUTPUT;
            END;

            FORMAT month_date DATE9.;
        run;

        /* Log progress */
        %PUT Processed score range &min_score to &max_score;
    %END;
%MEND;

/* Usage example */
%analyze_risk_segments(
    input_ds=credit_data,
    score_ranges=300-599|600-699|700-850,
    output_prefix=risk_segment
);
```

4. Error Handling in Loops

```
/* Macro with error handling */
%MACRO safe_loop_processing;
    %LET error_count = 0;

    %DO i = 1 %TO 5;
        /* Try-catch style processing */
        %LET rc = 0;

        DATA _NULL_;
            /* Some risky operation */
            IF &i = 3 THEN DO;
                rc = 1;
                CALL SYMPUTX('rc', rc);
            END;
        run;
```

```
        /* Check for errors */
        %IF &rc = 0 %THEN %DO;
            %PUT Processing successful for iteration &i;
        %END;
        %ELSE %DO;
            %LET error_count = %EVAL(&error_count + 1);
            %PUT ERROR: Processing failed for iteration &i;
        %END;
    %END;

    /* Final status */
    %PUT Processing completed with &error_count errors;
%MEND;
```

Best Practices:

1. Always initialize counter variables
2. Use appropriate loop type for the task
3. Include error handling
4. Avoid infinite loops
5. Document loop conditions and exit criteria
6. Consider performance for large iterations

Conditional blocks within data step and within Macros

1. DATA Step Conditional Blocks

A. Basic IF-THEN-ELSE

```
DATA credit_rating;
    SET customer_data;

    /* Simple IF-THEN */
    IF credit_score >= 750 THEN rating = 'Excellent';
    ELSE IF credit_score >= 700 THEN rating = 'Good';
    ELSE IF credit_score >= 650 THEN rating = 'Fair';
    ELSE rating = 'Poor';
```

```
    /* With compound conditions */
    IF credit_score >= 700 AND income > 50000 THEN
        risk_level = 'Low';
    ELSE IF credit_score >= 650 OR income > 75000 THEN
        risk_level = 'Medium';
    ELSE
        risk_level = 'High';
run;
```

B. IF-THEN-DO Blocks

```
DATA risk_analysis;
    SET loan_data;

    IF credit_score >= 700 THEN DO;
        risk_category = 'Low';
        interest_rate = 0.05;
        max_loan = income * 5;
        approval_flag = 1;
    END;
    ELSE DO;
        risk_category = 'High';
        interest_rate = 0.08;
        max_loan = income * 3;
        approval_flag = 0;
    END;
run;
```

C. SELECT Statements

```
DATA account_status;
    SET customer_accounts;

    /* Simple SELECT */
    SELECT;
        WHEN (balance < 0) status = 'Overdrawn';
        WHEN (balance = 0) status = 'Empty';
        WHEN (balance < 1000) status = 'Low';
        OTHERWISE status = 'Good';
    END;

    /* SELECT with compound conditions */
    SELECT;
        WHEN (balance >= 10000 AND months_active > 12)
            customer_tier = 'Premium';
        WHEN (balance >= 5000 OR months_active > 24)
            customer_tier = 'Gold';
```

```
        OTHERWISE
            customer_tier = 'Standard';
    END;
run;
```

D. Complex Nested Conditions

```
DATA loan_approval;
    SET applications;

    /* Multiple nested conditions */
    IF credit_score >= 700 THEN DO;
        IF income >= 75000 THEN DO;
            IF debt_ratio <= 0.3 THEN DO;
                approval_status = 'Approved';
                loan_limit = income * 5;
                interest_rate = 0.045;
            END;
            ELSE DO;
                approval_status = 'Conditional';
                loan_limit = income * 3;
                interest_rate = 0.055;
            END;
        END;
        ELSE DO;
            IF debt_ratio <= 0.25 THEN DO;
                approval_status = 'Conditional';
                loan_limit = income * 2.5;
                interest_rate = 0.065;
            END;
            ELSE DO;
                approval_status = 'Denied';
            END;
        END;
    END;
    ELSE DO;
        approval_status = 'Denied';
    END;
run;
```

2. Macro Conditional Blocks

A. Basic %IF-%THEN-%ELSE

```
%MACRO check_score(score=);
    %IF &score >= 750 %THEN %DO;
        %PUT Excellent credit score;
    %END;
    %ELSE %IF &score >= 700 %THEN %DO;
        %PUT Good credit score;
    %END;
    %ELSE %DO;
        %PUT Review required;
    %END;
%MEND;
```

B. Complex Macro Conditions

```
%MACRO process_data(
    dataset=,
    date=,
    type=
);
    /* Parameter validation */
    %IF %SYSFUNC(EXIST(&dataset)) %THEN %DO;
        %IF %SYSFUNC(INPUTN(&date, YYMMDD10.)) > 0 %THEN %DO;
            %IF %UPCASE(&type) IN (FULL PARTIAL SUMMARY) %THEN %DO;
                /* Process the data */
                DATA processed_&type;
                    SET &dataset;
                    WHERE date = "&date"d;
                run;
            %END;
            %ELSE %DO;
                %PUT ERROR: Invalid type specified;
            %END;
        %END;
        %ELSE %DO;
            %PUT ERROR: Invalid date format;
        %END;
    %END;
    %ELSE %DO;
        %PUT ERROR: Dataset does not exist;
    %END;
%MEND;
```

C. Conditional Macro Execution

```
%MACRO analyze_portfolio(
    run_risk=Y,
    run_revenue=Y,
    run_customer=Y
);
    %IF %UPCASE(&run_risk) = Y %THEN %DO;
        PROC SQL;
            CREATE TABLE risk_metrics AS
            SELECT customer_id,
                   credit_score,
                   default_probability
            FROM portfolio_data;
        QUIT;
    %END;

    %IF %UPCASE(&run_revenue) = Y %THEN %DO;
        PROC SQL;
            CREATE TABLE revenue_metrics AS
            SELECT customer_id,
                   SUM(transaction_amount) as total_revenue
            FROM transaction_data
            GROUP BY customer_id;
        QUIT;
    %END;

    %IF %UPCASE(&run_customer) = Y %THEN %DO;
        PROC SQL;
            CREATE TABLE customer_metrics AS
            SELECT customer_id,
                   COUNT(*) as transaction_count
            FROM transaction_data
            GROUP BY customer_id;
        QUIT;
    %END;
%MEND;
```

3. Combined Complex Examples

```
/* Complex credit risk assessment system */
%MACRO assess_credit_risk(
    input_ds=,
    score_threshold=,
    income_threshold=,
    output_ds=
);
    /* Validate parameters */
```

```
    %IF %SYSFUNC(EXIST(&input_ds)) %THEN %DO;
        %IF %DATATYP(&score_threshold) = NUMERIC AND
            %DATATYP(&income_threshold) = NUMERIC %THEN %DO;

            /* Create risk assessment */
            DATA &output_ds;
                SET &input_ds;

                /* Complex nested conditions */
                IF credit_score >= &score_threshold THEN DO;
                    IF income >= &income_threshold THEN DO;
                        risk_level = 'Low';
                        IF debt_ratio <= 0.3 THEN DO;
                            approval_status = 'Approved';
                            credit_limit = income * 4;
                        END;
                        ELSE DO;
                            approval_status = 'Review';
                            credit_limit = income * 2;
                        END;
                    END;
                    ELSE DO;
                        risk_level = 'Medium';
                        approval_status = 'Review';
                        credit_limit = income * 1.5;
                    END;
                END;
                ELSE DO;
                    risk_level = 'High';
                    approval_status = 'Denied';
                    credit_limit = 0;
                END;

                /* Additional risk factors */
                SELECT;
                    WHEN (months_employed < 12)
                        employment_risk = 'High';
                    WHEN (months_employed < 24)
                        employment_risk = 'Medium';
                    OTHERWISE
                        employment_risk = 'Low';
                END;
            run;

            /* Log processing results */
            PROC SQL NOPRINT;
                SELECT COUNT(*) INTO :record_count
                FROM &output_ds;
            QUIT;

            %PUT NOTE: Processed &record_count records;
```

```
        %END;
        %ELSE %DO;
            %PUT ERROR: Invalid threshold parameters;
        %END;
    %END;
    %ELSE %DO;
        %PUT ERROR: Input dataset does not exist;
    %END;
%MEND;
```

Best Practices:

1. **Clear Logic Structure**

```
/* Use indentation for readability */
%IF &condition1 %THEN %DO;
    %IF &condition2 %THEN %DO;
        /* Action */
    %END;
%END;
```

2. **Error Handling**

```
/* Always validate inputs */
%IF %SYSFUNC(MISSING(&parameter)) %THEN %DO;
    %PUT ERROR: Required parameter is missing;
    %RETURN;
%END;
```

3. **Documentation**

```
/* Document complex conditions */
IF (credit_score >= 700 AND  /* Good credit score */
    income >= 50000 AND      /* Sufficient income */
    debt_ratio <= 0.3        /* Acceptable debt ratio */
   ) THEN DO;
    /* Process approval */
END;
```

4. **Use Appropriate Conditional Structures**

- Use IF-THEN for simple conditions
- Use SELECT for multiple exclusive conditions

- Use nested IF-THEN for complex hierarchical logic

Read/Export from CSV and Excel, Create Inline Datasets, Delete Datasets

- **Read file from CSV or Excel**
- **Export file to CSV or Excel**
- **Create a datset with datapoints in-place**
- **Delete datasets**

1. Reading Files

A. Reading CSV Files

```
/* Using PROC IMPORT */
PROC IMPORT
    DATAFILE="/path/to/file.csv"
    OUT=work.my_data
    DBMS=CSV
    REPLACE;
    GETNAMES=YES;  /* First row contains headers */
    GUESSINGROWS=MAX;  /* Use all rows for type detection */
run;

/* Using DATA Step */
DATA my_data;
    INFILE "/path/to/file.csv"
           DLM=','
           FIRSTOBS=2  /* Skip header row */
           DSD        /* Handle embedded delimiters */
           MISSOVER;  /* Handle missing values */
    INPUT
        customer_id $
        name $
        credit_score
        balance;
run;
```

B. Reading Excel Files

```
/* Using PROC IMPORT */
PROC IMPORT
    DATAFILE="/path/to/file.xlsx"
    OUT=work.excel_data
    DBMS=XLSX
    REPLACE;
    SHEET="Sheet1";  /* Specify sheet name */
    GETNAMES=YES;
    MIXED=YES;      /* Handle mixed data types */
run;

/* Using Excel Engine */
LIBNAME xl XLSX "/path/to/file.xlsx";

DATA my_data;
    SET xl."Sheet1$"n;
run;

LIBNAME xl CLEAR;
```

2. Exporting Files

A. Exporting to CSV

```
/* Using PROC EXPORT */
PROC EXPORT
    DATA=work.my_data
    OUTFILE="/path/to/output.csv"
    DBMS=CSV
    REPLACE;
run;

/* Using DATA Step */
DATA _NULL_;
    SET my_data;
    FILE "/path/to/output.csv" DLM=',' DSD;

    /* Write header on first row */
    IF _N_ = 1 THEN DO;
        PUT "Customer_ID,Name,Credit_Score,Balance";
    END;

    PUT customer_id name credit_score balance;
run;
```

B. Exporting to Excel

```
/* Using PROC EXPORT */
PROC EXPORT
    DATA=work.my_data
    OUTFILE="/path/to/output.xlsx"
    DBMS=XLSX
    REPLACE;
    SHEET="Report";
run;

/* Using Excel Engine */
LIBNAME xl XLSX "/path/to/output.xlsx";

DATA xl.Report;
    SET my_data;
run;

LIBNAME xl CLEAR;
```

3. Creating Datasets In-Place

A. Simple Dataset Creation

```
/* Method 1: Using DATALINES */
DATA customer_data;
    INPUT customer_id $ name $ credit_score balance;
    DATALINES;
001 John 720 5000
002 Mary 680 3000
003 Bob  750 7000
;
run;

/* Method 2: Using CARDS Statement */
DATA customer_data;
    INPUT customer_id $ name $ credit_score balance;
    CARDS;
001 John 720 5000
002 Mary 680 3000
003 Bob  750 7000
;
run;

/* Method 3: Using Assignment Statements */
DATA customer_data;
    LENGTH customer_id $3 name $20;
```

```
    customer_id = '001';
    name = 'John';
    credit_score = 720;
    balance = 5000;
    OUTPUT;

    customer_id = '002';
    name = 'Mary';
    credit_score = 680;
    balance = 3000;
    OUTPUT;
run;
```

B. Creating Empty Dataset with Structure

```
/* Create empty dataset with defined structure */
DATA empty_dataset;
    LENGTH
        customer_id $10
        name $50
        credit_score 8
        balance 8
        status $10;
    STOP;
run;
```

4. Deleting Datasets

A. Using PROC DELETE

```
/* Delete single dataset */
PROC DELETE DATA=work.my_data;
run;

/* Delete multiple datasets */
PROC DELETE DATA=work.dataset1 work.dataset2 work.dataset3;
run;
```

B. Using PROC DATASETS

```
/* Delete specific datasets */
PROC DATASETS LIBRARY=work NOLIST;
    DELETE dataset1 dataset2 dataset3;
QUIT;

/* Delete all datasets in a library */
PROC DATASETS LIBRARY=work KILL NOLIST;
QUIT;

/* Delete datasets matching pattern */
PROC DATASETS LIBRARY=work NOLIST;
    DELETE temp: report_:;  /* Deletes datasets starting with 'temp' and
'report_' */
QUIT;
```

5. Complex Examples

A. Comprehensive File Processing

```
/* Read, Process, and Export Data */
%MACRO process_customer_data(
    input_file=,
    output_file=
);
    /* Step 1: Import data with error handling */
    %LET import_success = 0;

    PROC IMPORT
        DATAFILE="&input_file"
        OUT=work.raw_data
        DBMS=CSV
        REPLACE;
        GETNAMES=YES;
        GUESSINGROWS=MAX;
    run;

    %IF &SYSERR = 0 %THEN %DO;
        %LET import_success = 1;

        /* Step 2: Clean and process data */
        DATA work.processed_data;
            SET work.raw_data;

            /* Clean and standardize data */
            customer_id = UPCASE(STRIP(customer_id));
```

```
            name = PROPCASE(STRIP(name));

            /* Validate numeric fields */
            IF NOT MISSING(credit_score) AND
               300 <= credit_score <= 850;

            /* Add processing metadata */
            process_date = TODAY();
            process_time = TIME();

            FORMAT
                process_date DATE9.
                process_time TIME8.
                balance DOLLAR12.2;
        run;

        /* Step 3: Export processed data */
        PROC EXPORT
            DATA=work.processed_data
            OUTFILE="&output_file"
            DBMS=XLSX
            REPLACE;
            SHEET="Processed_Data";
        run;

        /* Step 4: Cleanup temporary datasets */
        PROC DATASETS LIBRARY=work NOLIST;
            DELETE raw_data;
        QUIT;
    %END;
    %ELSE %DO;
        %PUT ERROR: Failed to import data from &input_file;
    %END;
%MEND;
```

6. Best Practices

1. **Error Handling**

```
/* Check if file exists before reading */
%LET fileref = myfile;
%LET rc = %SYSFUNC(FILENAME(fileref, &file_path));
%IF %SYSFUNC(FEXIST(&fileref)) %THEN %DO;
    /* Process file */
%END;
%ELSE %DO;
    %PUT ERROR: File does not exist;
%END;
```

2. **Data Validation**

```
/* Validate data during import */
DATA validated_data;
    SET imported_data;

    /* Check required fields */
    IF MISSING(customer_id) THEN DELETE;

    /* Validate numeric ranges */
    IF NOT(300 <= credit_score <= 850) THEN
        credit_score = .;

    /* Log invalid records */
    IF _ERROR_ THEN
        PUT "ERROR: Invalid record - " _ALL_;
run;
```

3. **Performance Optimization**

```
/* Use appropriate options for large files */
OPTIONS COMPRESS=YES;
PROC IMPORT
    DATAFILE="large_file.csv"
    OUT=work.large_data
    DBMS=CSV
    REPLACE;
    GETNAMES=YES;
    GUESSINGROWS=1000;  /* Limit rows for type detection */
run;
```

PROC SUMMARY, PROC FREQ, PROC FORMATS

1. PROC SUMMARY

```
/* Basic PROC SUMMARY */
PROC SUMMARY DATA=credit_data NWAY MISSING;
    CLASS customer_type region;
    VAR balance credit_score;
    OUTPUT OUT=summary_stats
        SUM=total_balance total_score
```

```
        MEAN=avg_balance avg_score
        MIN=min_balance min_score
        MAX=max_balance max_score
        N=count;
run;

/* Complex Example with Multiple Statistics */
PROC SUMMARY DATA=transaction_data
    NWAY       /* Unique combinations of CLASS variables */
    MISSING    /* Include missing values */
    COMPLETETYPES;  /* All possible combinations */

    /* Classification variables */
    CLASS
        year
        month
        region
        product_type;

    /* Analysis variables */
    VAR
        transaction_amount
        customer_count
        profit_margin;

    /* Output specifications */
    OUTPUT OUT=detailed_summary(DROP=_TYPE_ _FREQ_)
        /* Basic statistics */
        N=record_count
        NMISS=missing_count
        SUM=total_amount total_customers total_profit
        MEAN=avg_amount avg_customers avg_profit

        /* Spread statistics */
        STD=std_amount std_customers std_profit
        VAR=var_amount var_customers var_profit

        /* Range statistics */
        MIN=min_amount min_customers min_profit
        MAX=max_amount max_customers max_profit
        RANGE=range_amount range_customers range_profit

        /* Percentiles */
        P1=p1_amount
        P5=p5_amount
        P10=p10_amount
        P90=p90_amount
        P95=p95_amount
        P99=p99_amount;
run;
```

2. PROC FREQ

```
/* Basic Frequency Analysis */
PROC FREQ DATA=credit_data;
    TABLES credit_rating status;
run;

/* Cross-tabulation */
PROC FREQ DATA=credit_data;
    TABLES credit_rating * status /
        NOCOL NOROW NOPERCENT MISSING;
run;

/* Complex PROC FREQ with Statistics */
PROC FREQ DATA=loan_data;
    /* Define formats for grouping */
    FORMAT
        credit_score credit_groups.
        income income_groups.
        age age_groups.;

    /* Complex table specifications */
    TABLES
        /* Two-way tables */
        (credit_score income age) * loan_status /
            CHISQ      /* Chi-square test */
            MEASURES   /* Association measures */
            NOCOL      /* Suppress column percentages */
            PLOTS=ALL  /* All available plots */
            OUT=freq_out;

    /* Three-way tables */
    TABLES region * credit_score * loan_status /
        CHISQ
        EXACT      /* Exact tests */
        EXPECTED   /* Expected frequencies */
        DEVIATION  /* Deviations from expected */
        NOPRINT    /* Suppress printed output */
        OUT=region_analysis;

    /* Output statistics */
    OUTPUT
        OUT=chi_square_stats
        CHISQ;

    /* Weight cases */
    WEIGHT amount;

    /* By group processing */
```

```
    BY year quarter;
run;
```

3. PROC FORMAT

```
/* Basic Format Creation */
PROC FORMAT;
    /* Numeric ranges */
    VALUE credit_score_fmt
        LOW-649 = 'Poor'
        650-699 = 'Fair'
        700-749 = 'Good'
        750-HIGH = 'Excellent';

    /* Character values */
    VALUE $status_fmt
        'A' = 'Active'
        'I' = 'Inactive'
        'P' = 'Pending'
        OTHER = 'Unknown';
run;

/* Complex Format Examples */
PROC FORMAT;
    /* Numeric ranges with overlaps */
    VALUE age_fmt
        LOW-17 = 'Under 18'
        18-24 = 'Young Adult'
        25-34 = 'Adult'
        35-49 = 'Middle Age'
        50-64 = 'Senior'
        65-HIGH = 'Retired'
        . = 'Unknown';

    /* Character format with special handling */
    VALUE $region_fmt (DEFAULT=12)
        'NORTH' = 'Northern Region'
        'SOUTH' = 'Southern Region'
        'EAST' = 'Eastern Region'
        'WEST' = 'Western Region'
        OTHER = 'Unknown Region';

    /* Multilabel format */
    VALUE risk_fmt (MULTILABEL)
        LOW-599 = 'High Risk'
        600-699 = 'Medium Risk'
        700-HIGH = 'Low Risk'
        LOW-699 = 'Review Required'
```

```
        700-HIGH = 'Auto Approve';

    /* Picture format for custom display */
    PICTURE phone_fmt
        OTHER = '(999)999-9999' (PREFIX='(');
run;

/* Using Formats in Analysis */
DATA credit_analysis;
    SET credit_data;

    /* Apply formats */
    FORMAT
        credit_score credit_score_fmt.
        status $status_fmt.
        age age_fmt.
        region $region_fmt.
        risk_level risk_fmt.
        phone phone_fmt.;
run;
```

4. Combined Complex Example

```
/* Comprehensive Credit Risk Analysis */

/* Step 1: Create Custom Formats */
PROC FORMAT;
    /* Credit Score Categories */
    VALUE score_fmt
        LOW-649 = 'High Risk'
        650-699 = 'Medium Risk'
        700-749 = 'Low Risk'
        750-HIGH = 'Prime';

    /* Income Brackets */
    VALUE income_fmt (MULTILABEL)
        LOW-29999 = 'Low Income'
        30000-59999 = 'Middle Income'
        60000-99999 = 'Upper Middle'
        100000-HIGH = 'High Income'
        LOW-59999 = 'Standard Products'
        60000-HIGH = 'Premium Products';

    /* Age Groups */
    VALUE age_fmt
        LOW-25 = 'Young'
        26-40 = 'Adult'
        41-60 = 'Middle Age'
```

```
        61-HIGH = 'Senior';
run;

/* Step 2: Detailed Summary Analysis */
PROC SUMMARY DATA=credit_data NWAY MISSING;
    /* Apply formats */
    FORMAT
        credit_score score_fmt.
        annual_income income_fmt.
        age age_fmt.;

    /* Classification variables */
    CLASS
        credit_score
        annual_income
        age
        region;

    /* Analysis variables */
    VAR
        loan_amount
        interest_rate
        default_risk;

    /* Output detailed statistics */
    OUTPUT OUT=risk_summary(DROP=_TYPE_ _FREQ_)
        N=count
        SUM=total_loan total_interest total_risk
        MEAN=avg_loan avg_interest avg_risk
        STD=std_loan std_interest std_risk
        MIN=min_loan min_interest min_risk
        MAX=max_loan max_interest max_risk;
run;

/* Step 3: Frequency Analysis */
PROC FREQ DATA=credit_data;
    /* Apply formats */
    FORMAT
        credit_score score_fmt.
        annual_income income_fmt.
        age age_fmt.;

    /* Complex cross-tabulations */
    TABLES
        (credit_score income age) *
        (loan_status default_flag) /
            CHISQ
            MEASURES
            PLOTS=ALL
            OUT=freq_analysis;
```

```
    /* Output statistics */
    OUTPUT
        OUT=chi_square_results
        CHISQ;

    /* Weight by loan amount */
    WEIGHT loan_amount;
run;

/* Step 4: Generate Summary Report */
PROC PRINT DATA=risk_summary NOOBS;
    BY region;
    SUM count total_loan total_interest;
    FORMAT
        total_loan total_interest DOLLAR12.2
        avg_loan avg_interest DOLLAR8.2
        std_loan std_interest DOLLAR8.2;
run;
```

Best Practices:

1. **Use Appropriate Options**

```
/* PROC SUMMARY */
PROC SUMMARY DATA=mydata
    NWAY       /* Efficient for unique combinations */
    MISSING    /* Include missing values */
    COMPLETETYPES;  /* All possible combinations */

/* PROC FREQ */
PROC FREQ DATA=mydata
    ORDER=FREQ  /* Order by frequency */
    MISSING;    /* Include missing values */

/* PROC FORMAT */
PROC FORMAT
    LIBRARY=work  /* Temporary formats */
    FMTLIB;      /* List format definitions */
```

2. **Handle Missing Values**

```
/* Include missing value categories */
PROC FORMAT;
    VALUE score_fmt
        LOW-649 = 'High Risk'
        650-HIGH = 'Low Risk'
        . = 'Unknown';
run;
```

3. **Document Format Definitions**

```
PROC FORMAT;
    VALUE age_fmt (DEFAULT=12)  /* Default length */
        /* Age categories for risk analysis */
        LOW-25 = 'Young'     /* Higher risk */
        26-40 = 'Adult'      /* Moderate risk */
        41-HIGH = 'Senior'   /* Lower risk */
        . = 'Unknown';       /* Requires review */
run;
```

That's all folks 🎉✨