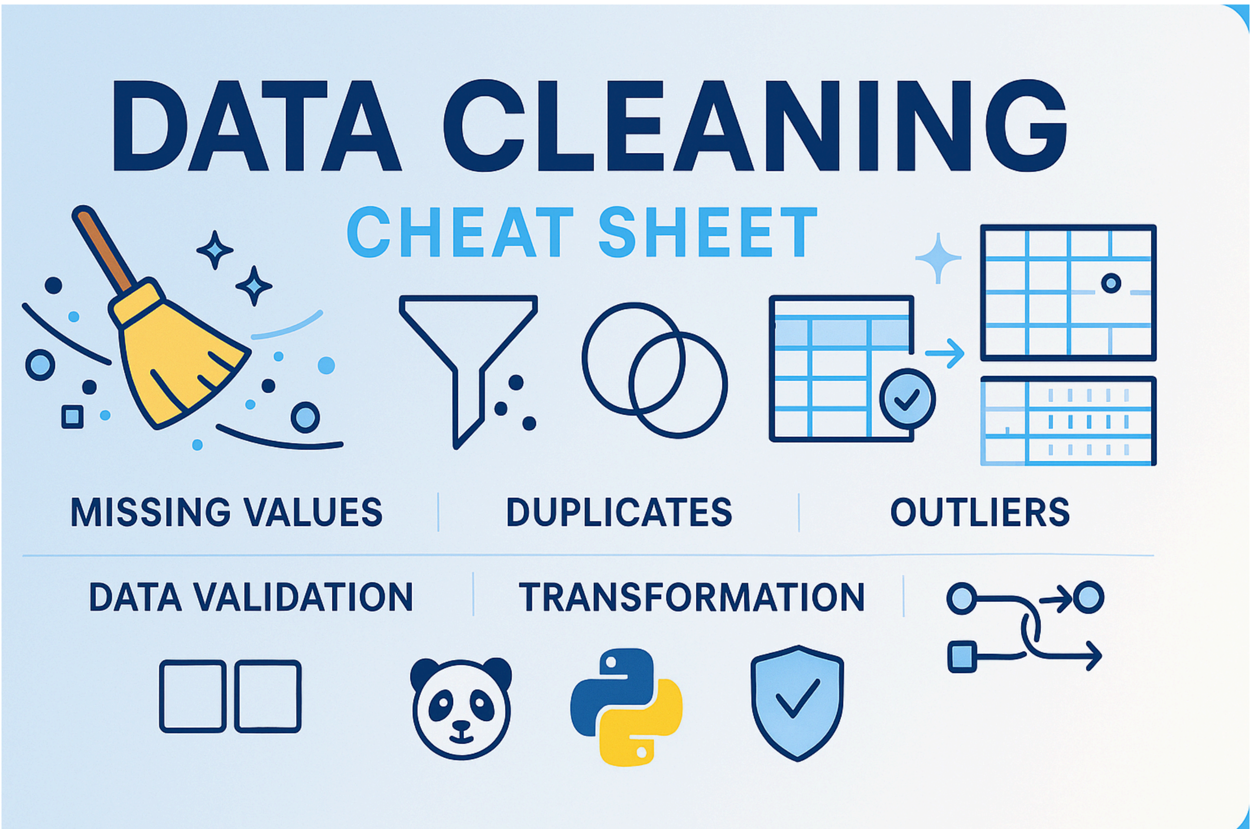
Data Cleaning - Comprehensive Cheat Sheet



Data Quality Assessment

Common Data Quality Issues

```
# Missing Data
- Null values (None, NaN, NULL)
- Empty strings ("")
- Placeholder values ("N/A", "Unknown", -999)
- Inconsistent missing representations

# Inconsistent Data
```

```
- Different formats for same data
- Varying case (uppercase/lowercase)
- Different date formats
- Unit inconsistencies

# Invalid Data
- Out-of-range values
- Impossible combinations
- Wrong data types
- Duplicate records

# Structural Issues
- Incorrect column names
- Wrong data types
- Inconsistent schemas
- Encoding problems
```

Data Profiling

```
import pandas as pd
import numpy as np

# Basic dataset overview
df.info()                  # Data types, non-null counts
df.describe()              # Statistical summary
df.shape                   # Dimensions
df.columns.tolist()        # Column names
df.dtypes                  # Data types

# Missing data analysis
df.isnull().sum()          # Count missing values per column
df.isnull().sum() / len(df)  # Percentage missing
df.isnull().any(axis=1).sum() # Rows with any missing values

# Unique values analysis
```

```
df.nunique()                # Unique values per column
df['column'].value_counts()  # Frequency distribution
df.duplicated().sum()       # Count duplicate rows

# Data range analysis
df.select_dtypes(include=[np.number]).describe()  # Numeric summaries
df.select_dtypes(include=['object']).describe()   # Text summaries
```

Missing Data Handling

Types of Missing Data

```
# Missing Completely at Random (MCAR)
- Missing values are random
- No pattern in missingness
- Safe to delete or impute

# Missing at Random (MAR)
- Missing depends on observed data
- Can be predicted from other variables
- Use advanced imputation methods

# Missing Not at Random (MNAR)
- Missing depends on unobserved data
- Missingness is systematic
- Requires domain knowledge
```

Missing Data Detection

```
# Visualize missing patterns
import seaborn as sns
import matplotlib.pyplot as plt

# Missing data heatmap
```

```
plt.figure(figsize=(12, 8))
sns.heatmap(df.isnull(), cbar=True, yticklabels=False)

# Missing data bar chart
missing_data = df.isnull().sum()
missing_data = missing_data[missing_data > 0].sort_values(ascending=False)
plt.figure(figsize=(10, 6))
missing_data.plot(kind='bar')

# Missing data patterns
import missingno as msno
msno.matrix(df)          # Missing data matrix
msno.bar(df)             # Missing data bar chart
msno.heatmap(df)         # Missing data correlation heatmap
msno.dendrogram(df)      # Missing data dendrogram
```

Missing Data Treatment

Deletion Methods

```
# Listwise deletion (remove rows with any missing)
df_clean = df.dropna()

# Pairwise deletion (use available data for each analysis)
correlation = df.corr()    # Automatically handles missing values

# Remove rows with missing in specific columns
df_clean = df.dropna(subset=['important_column'])

# Remove rows with more than X missing values
threshold = len(df.columns) * 0.5  # Keep rows with <50% missing
df_clean = df.dropna(thresh=threshold)

# Remove columns with high missing percentage
```

```
threshold = 0.7  # Remove columns with >70% missing
df_clean = df.loc[:, df.isnull().mean() < threshold]
```

Imputation Methods

```
# Simple imputation
df['column'].fillna(df['column'].mean())     # Mean imputation
df['column'].fillna(df['column'].median())   # Median imputation
df['column'].fillna(df['column'].mode()[0])  # Mode imputation
df['column'].fillna(method='ffill')          # Forward fill
df['column'].fillna(method='bfill')          # Backward fill

# Group-based imputation
df['column'] = df.groupby('category')['column'].transform(
    lambda x: x.fillna(x.mean())
)

# Interpolation for time series
df['column'] = df['column'].interpolate(method='linear')
df['column'] = df['column'].interpolate(method='polynomial', order=2)

# Advanced imputation with sklearn
from sklearn.impute import SimpleImputer, KNNImputer, IterativeImputer

# Simple imputer
imputer = SimpleImputer(strategy='mean')  # 'median', 'most_frequent', 'const
ant'
df_imputed = pd.DataFrame(imputer.fit_transform(df), columns=df.columns)

# KNN imputation
knn_imputer = KNNImputer(n_neighbors=5)
df_imputed = pd.DataFrame(knn_imputer.fit_transform(df), columns=df.colum
ns)

# Iterative imputation (MICE)
```

```
iter_imputer = IterativeImputer(random_state=42)
df_imputed = pd.DataFrame(iter_imputer.fit_transform(df), columns=df.column
s)
```

Duplicate Data Handling

Duplicate Detection

```
# Find duplicate rows
duplicates = df.duplicated()             # Boolean mask
duplicate_rows = df[df.duplicated()]     # Show duplicate rows
df.duplicated().sum()                    # Count duplicates

# Find duplicates based on specific columns
df.duplicated(subset=['col1', 'col2'])

# Find duplicates keeping different occurrences
df.duplicated(keep='first')   # Mark all but first as duplicate
df.duplicated(keep='last')    # Mark all but last as duplicate
df.duplicated(keep=False)     # Mark all duplicates
```

Duplicate Removal

```
# Remove duplicate rows
df_clean = df.drop_duplicates()

# Remove duplicates based on specific columns
df_clean = df.drop_duplicates(subset=['col1', 'col2'])

# Keep specific duplicate
df_clean = df.drop_duplicates(keep='last')  # Keep last occurrence

# Advanced duplicate handling
# Keep the row with most complete data
```

```
def keep_most_complete(group):
    return group.loc[group.isnull().sum(axis=1).idxmin()]

df_clean = df.groupby(['id_column']).apply(keep_most_complete).reset_index
(drop=True)
```

Data Type Conversion

Numeric Conversions

```
# Convert to numeric
df['column'] = pd.to_numeric(df['column'], errors='coerce')  # NaN for invalid
df['column'] = pd.to_numeric(df['column'], errors='ignore')  # Keep original if i
nvalid

# Handle specific formats
df['price'] = df['price'].str.replace('$', '').str.replace(',', '').astype(float)
df['percentage'] = df['percentage'].str.rstrip('%').astype(float) / 100

# Convert scientific notation
df['scientific'] = df['scientific'].astype(float)

# Handle negative numbers in parentheses
df['amount'] = df['amount'].str.replace(r'\((.*)\)', r'-\1', regex=True).astype(floa
t)
```

Date/Time Conversions

```
# Basic date conversion
df['date'] = pd.to_datetime(df['date'])

# Handle different date formats
df['date'] = pd.to_datetime(df['date'], format='%Y-%m-%d')
df['date'] = pd.to_datetime(df['date'], format='%d/%m/%Y')
```

```
df['date'] = pd.to_datetime(df['date'], infer_datetime_format=True)

# Handle errors in date conversion
df['date'] = pd.to_datetime(df['date'], errors='coerce')  # NaT for invalid

# Extract date components
df['year'] = df['date'].dt.year
df['month'] = df['date'].dt.month
df['day'] = df['date'].dt.day
df['weekday'] = df['date'].dt.day_name()
df['quarter'] = df['date'].dt.quarter

# Handle timezone
df['date'] = pd.to_datetime(df['date'], utc=True)
df['date'] = df['date'].dt.tz_convert('US/Eastern')
```

String Conversions

```
# Basic string operations
df['column'] = df['column'].astype(str)
df['column'] = df['column'].str.strip()      # Remove whitespace
df['column'] = df['column'].str.lower()      # Lowercase
df['column'] = df['column'].str.upper()      # Uppercase
df['column'] = df['column'].str.title()      # Title case

# Handle encoding issues
df['column'] = df['column'].str.encode('utf-8').str.decode('utf-8')

# Replace special characters
df['column'] = df['column'].str.replace('[^\w\s]', '', regex=True)  # Keep only alphanumeric
```

Categorical Conversions

```
# Convert to categorical
df['category'] = df['category'].astype('category')

# Ordered categorical
df['size'] = pd.Categorical(df['size'],
                            categories=['Small', 'Medium', 'Large'],
                            ordered=True)

# Map values to categories
category_map = {'A': 'Group1', 'B': 'Group2', 'C': 'Group1'}
df['new_category'] = df['old_category'].map(category_map)
```

Outlier Detection and Treatment

Statistical Methods

```
# Z-score method
from scipy import stats
z_scores = np.abs(stats.zscore(df['column']))
outliers = df[z_scores > 3]  # Values more than 3 standard deviations

# IQR method
Q1 = df['column'].quantile(0.25)
Q3 = df['column'].quantile(0.75)
IQR = Q3 - Q1
lower_bound = Q1 - 1.5 * IQR
upper_bound = Q3 + 1.5 * IQR
outliers = df[(df['column'] < lower_bound) | (df['column'] > upper_bound)]

# Modified Z-score (robust)
median = df['column'].median()
mad = np.median(np.abs(df['column'] - median))
modified_z_scores = 0.6745 * (df['column'] - median) / mad
outliers = df[np.abs(modified_z_scores) > 3.5]
```

```
# Percentile method
lower_percentile = df['column'].quantile(0.01)
upper_percentile = df['column'].quantile(0.99)
outliers = df[(df['column'] < lower_percentile) | (df['column'] > upper_percenti
le)]
```

Machine Learning Methods

```
from sklearn.ensemble import IsolationForest
from sklearn.neighbors import LocalOutlierFactor
from sklearn.svm import OneClassSVM

# Isolation Forest
iso_forest = IsolationForest(contamination=0.1, random_state=42)
outliers = iso_forest.fit_predict(df[numeric_columns])
df['outlier'] = outliers

# Local Outlier Factor
lof = LocalOutlierFactor(n_neighbors=20, contamination=0.1)
outliers = lof.fit_predict(df[numeric_columns])

# One-Class SVM
svm = OneClassSVM(nu=0.1)
outliers = svm.fit_predict(df[numeric_columns])
```

Outlier Treatment

```
# Remove outliers
df_clean = df[~df['outlier_flag']]

# Cap outliers (winsorization)
df['column_capped'] = df['column'].clip(lower=lower_bound, upper=upper_bo
und)
```

```
# Transform outliers
df['column_log'] = np.log1p(df['column'])  # Log transformation
df['column_sqrt'] = np.sqrt(df['column'])  # Square root transformation

# Replace with median/mean
outlier_mask = (df['column'] < lower_bound) | (df['column'] > upper_bound)
df.loc[outlier_mask, 'column'] = df['column'].median()
```

Text Data Cleaning

Basic Text Cleaning

```
import re
import string

# Remove punctuation
df['text'] = df['text'].str.translate(str.maketrans('', '', string.punctuation))

# Remove numbers
df['text'] = df['text'].str.replace('\d+', '', regex=True)

# Remove extra whitespace
df['text'] = df['text'].str.strip().str.replace('\s+', ' ', regex=True)

# Remove special characters
df['text'] = df['text'].str.replace('[^a-zA-Z\s]', '', regex=True)

# Handle case sensitivity
df['text'] = df['text'].str.lower()

# Remove HTML tags
df['text'] = df['text'].str.replace('<.*?>', '', regex=True)

# Remove URLs
df['text'] = df['text'].str.replace(r'http\S+|www\S+|https\S+', '', regex=True)
```

```
# Remove email addresses
df['text'] = df['text'].str.replace(r'\S+@\S+', '', regex=True)
```

Advanced Text Processing

```
import nltk
from nltk.corpus import stopwords
from nltk.stem import PorterStemmer, WordNetLemmatizer
from nltk.tokenize import word_tokenize

# Download required NLTK data
nltk.download('stopwords')
nltk.download('punkt')
nltk.download('wordnet')

# Remove stopwords
stop_words = set(stopwords.words('english'))
df['text'] = df['text'].apply(lambda x: ' '.join([word for word in x.split() if word.l
ower() not in stop_words]))

# Stemming
stemmer = PorterStemmer()
df['text_stemmed'] = df['text'].apply(lambda x: ' '.join([stemmer.stem(word) fo
r word in x.split()]))

# Lemmatization
lemmatizer = WordNetLemmatizer()
df['text_lemmatized'] = df['text'].apply(lambda x: ' '.join([lemmatizer.lemmatiz
e(word) for word in x.split()]))

# Tokenization
df['tokens'] = df['text'].apply(word_tokenize)

# Remove short words
```

```
df['text'] = df['text'].apply(lambda x: ' '.join([word for word in x.split() if len(wo
rd) > 2]))
```

Data Standardization

Format Standardization

```
# Standardize phone numbers
def standardize_phone(phone):
    # Remove all non-digits
    digits = re.sub(r'\D', '', str(phone))
    if len(digits) == 10:
        return f"({digits[:3]}) {digits[3:6]}-{digits[6:]}"
    return phone

df['phone'] = df['phone'].apply(standardize_phone)

# Standardize addresses
def standardize_address(address):
    address = str(address).upper()
    # Replace common abbreviations
    replacements = {
        'STREET': 'ST', 'AVENUE': 'AVE', 'BOULEVARD': 'BLVD',
        'DRIVE': 'DR', 'COURT': 'CT', 'PLACE': 'PL'
    }
    for full, abbrev in replacements.items():
        address = address.replace(full, abbrev)
    return address

df['address'] = df['address'].apply(standardize_address)

# Standardize country names
country_mapping = {
    'USA': 'United States', 'US': 'United States',
    'UK': 'United Kingdom', 'GB': 'United Kingdom'
```

```
}
df['country'] = df['country'].replace(country_mapping)
```

Value Standardization

```
# Standardize boolean values
boolean_map = {
    'yes': True, 'no': False, 'y': True, 'n': False,
    '1': True, '0': False, 'true': True, 'false': False
}
df['boolean_col'] = df['boolean_col'].str.lower().map(boolean_map)

# Standardize gender values
gender_map = {
    'm': 'Male', 'f': 'Female', 'male': 'Male', 'female': 'Female',
    'man': 'Male', 'woman': 'Female'
}
df['gender'] = df['gender'].str.lower().map(gender_map)

# Standardize units
def convert_units(value, unit):
    if unit.lower() in ['kg', 'kilogram']:
        return value  # Keep as kg
    elif unit.lower() in ['g', 'gram']:
        return value / 1000  # Convert to kg
    elif unit.lower() in ['lb', 'pound']:
        return value * 0.453592  # Convert to kg
    return value

df['weight_kg'] = df.apply(lambda row: convert_units(row['weight'], row['unit']), axis=1)
```

Data Validation

Range Validation

```
# Check numeric ranges
age_valid = df['age'].between(0, 120)
invalid_ages = df[~age_valid]

# Check date ranges
date_valid = df['date'].between('2020-01-01', '2023-12-31')
invalid_dates = df[~date_valid]

# Custom validation functions
def validate_email(email):
    pattern = r'^[a-zA-Z0-9._%+-]+@[a-zA-Z0-9.-]+\.[a-zA-Z]{2,}$'
    return bool(re.match(pattern, str(email)))

df['email_valid'] = df['email'].apply(validate_email)

def validate_credit_card(number):
    # Simple Luhn algorithm check
    number = str(number).replace(' ', '').replace('-', '')
    if not number.isdigit():
        return False

    def luhn_check(card_num):
        def digits_of(n):
            return [int(d) for d in str(n)]

        digits = digits_of(card_num)
        odd_digits = digits[-1::-2]
        even_digits = digits[-2::-2]
        checksum = sum(odd_digits)
        for d in even_digits:
            checksum += sum(digits_of(d*2))
        return checksum % 10 == 0

    return luhn_check(number)
```

```
df['cc_valid'] = df['credit_card'].apply(validate_credit_card)
```

Consistency Validation

```
# Check logical consistency
# Birth date should be before current date
df['birth_date_valid'] = df['birth_date'] < pd.Timestamp.now()

# Start date should be before end date
df['date_consistency'] = df['start_date'] < df['end_date']

# Price should be positive
df['price_valid'] = df['price'] >= 0

# Cross-field validation
def validate_age_birth_date(row):
    if pd.isna(row['age']) or pd.isna(row['birth_date']):
        return True  # Skip validation if either is missing

    calculated_age = (pd.Timestamp.now() - row['birth_date']).days // 365
    return abs(calculated_age - row['age']) <= 1  # Allow 1 year difference

df['age_birth_consistency'] = df.apply(validate_age_birth_date, axis=1)
```

Encoding and Transformation

Categorical Encoding

```
from sklearn.preprocessing import LabelEncoder, OneHotEncoder

# Label encoding (ordinal)
label_encoder = LabelEncoder()
df['category_encoded'] = label_encoder.fit_transform(df['category'])
```

```
# One-hot encoding
df_encoded = pd.get_dummies(df, columns=['category'], prefix='cat')

# Or using sklearn
one_hot = OneHotEncoder(sparse=False, drop='first')
encoded_array = one_hot.fit_transform(df[['category']])
encoded_df = pd.DataFrame(encoded_array, columns=one_hot.get_feature_n
ames_out())

# Target encoding (mean encoding)
mean_encoded = df.groupby('category')['target'].mean()
df['category_target_encoded'] = df['category'].map(mean_encoded)
```

Numerical Scaling

```
from sklearn.preprocessing import StandardScaler, MinMaxScaler, RobustScal
er

# Standard scaling (z-score normalization)
scaler = StandardScaler()
df['scaled'] = scaler.fit_transform(df[['column']])

# Min-max scaling
minmax_scaler = MinMaxScaler()
df['normalized'] = minmax_scaler.fit_transform(df[['column']])

# Robust scaling (less sensitive to outliers)
robust_scaler = RobustScaler()
df['robust_scaled'] = robust_scaler.fit_transform(df[['column']])

# Manual scaling
df['manual_scaled'] = (df['column'] - df['column'].mean()) / df['column'].std()
```

Data Quality Reporting

Quality Metrics

```
def data_quality_report(df):
    report = {}

    # Basic info
    report['total_rows'] = len(df)
    report['total_columns'] = len(df.columns)

    # Missing data
    report['missing_data'] = {
        'total_missing_values': df.isnull().sum().sum(),
        'columns_with_missing': df.isnull().any().sum(),
        'rows_with_missing': df.isnull().any(axis=1).sum(),
        'missing_percentage': (df.isnull().sum().sum() / (len(df) * len(df.column
s))) * 100
    }

    # Duplicates
    report['duplicates'] = {
        'duplicate_rows': df.duplicated().sum(),
        'duplicate_percentage': (df.duplicated().sum() / len(df)) * 100
    }

    # Data types
    report['data_types'] = df.dtypes.value_counts().to_dict()

    # Unique values
    report['unique_values'] = df.nunique().describe().to_dict()

    return report

# Generate report
```

```
quality_report = data_quality_report(df)
print(quality_report)
```

Validation Report

```
def create_validation_report(df, validation_rules):
    """
    Create a validation report based on predefined rules

    validation_rules: dict with column names as keys and validation functions as
values
    """
    report = {}

    for column, rule in validation_rules.items():
        if column in df.columns:
            try:
                valid_mask = df[column].apply(rule)
                report[column] = {
                    'total_values': len(df[column]),
                    'valid_values': valid_mask.sum(),
                    'invalid_values': (~valid_mask).sum(),
                    'validity_percentage': (valid_mask.sum() / len(df[column])) * 100,
                    'invalid_samples': df[~valid_mask][column].head().tolist()
                }
            except Exception as e:
                report[column] = {'error': str(e)}

    return report

# Define validation rules
validation_rules = {
    'age': lambda x: 0 <= x <= 120 if pd.notna(x) else True,
    'email': lambda x: '@' in str(x) if pd.notna(x) else True,
    'price': lambda x: x >= 0 if pd.notna(x) else True
```

```
}

validation_report = create_validation_report(df, validation_rules)
```

Automated Data Cleaning Pipeline

Pipeline Template

```
class DataCleaningPipeline:
    def __init__(self):
        self.steps = []
        self.transformers = {}

    def add_step(self, step_name, step_function):
        self.steps.append((step_name, step_function))

    def fit_transform(self, df):
        cleaned_df = df.copy()

        for step_name, step_function in self.steps:
            print(f"Executing step: {step_name}")
            cleaned_df = step_function(cleaned_df)
            print(f"Data shape after {step_name}: {cleaned_df.shape}")

        return cleaned_df

    def transform(self, df):
        # Apply fitted transformations to new data
        return self.fit_transform(df)

# Example usage
def remove_duplicates(df):
    return df.drop_duplicates()

def handle_missing_values(df):
```

```
    # Simple imputation strategy
    numeric_columns = df.select_dtypes(include=[np.number]).columns
    categorical_columns = df.select_dtypes(include=['object']).columns

    df[numeric_columns] = df[numeric_columns].fillna(df[numeric_columns].m
edian())
    df[categorical_columns] = df[categorical_columns].fillna(df[categorical_col
umns].mode().iloc[0])

    return df

def standardize_text(df):
    text_columns = df.select_dtypes(include=['object']).columns
    for col in text_columns:
        df[col] = df[col].str.strip().str.lower()
    return df

# Build pipeline
pipeline = DataCleaningPipeline()
pipeline.add_step("Remove Duplicates", remove_duplicates)
pipeline.add_step("Handle Missing Values", handle_missing_values)
pipeline.add_step("Standardize Text", standardize_text)

# Execute pipeline
cleaned_df = pipeline.fit_transform(df)
```

Best Practices

Documentation and Tracking

```
# Track cleaning operations
cleaning_log = []

def log_operation(operation, before_shape, after_shape, details=""):
    cleaning_log.append({
```

```
        'operation': operation,
        'before_shape': before_shape,
        'after_shape': after_shape,
        'rows_affected': before_shape[0] - after_shape[0],
        'details': details,
        'timestamp': pd.Timestamp.now()
    })

# Example usage
before_shape = df.shape
df_clean = df.dropna()
after_shape = df_clean.shape
log_operation("Remove missing values", before_shape, after_shape)

# Convert log to DataFrame
cleaning_summary = pd.DataFrame(cleaning_log)
```

Data Backup and Versioning

```
# Save original data
df_original = df.copy()

# Save intermediate steps
df.to_pickle('data_step1_duplicates_removed.pkl')
df.to_pickle('data_step2_missing_handled.pkl')

# Create data version metadata
metadata = {
    'version': '1.0',
    'date_processed': pd.Timestamp.now(),
    'operations_performed': [step[0] for step in pipeline.steps],
    'original_shape': df_original.shape,
    'final_shape': df.shape,
    'data_quality_score': calculate_quality_score(df)
}
```

```
# Save metadata
import json
with open('data_metadata.json', 'w') as f:
    json.dump(metadata, f, default=str)
```

Testing and Validation

```
# Unit tests for cleaning functions
def test_remove_duplicates():
    test_df = pd.DataFrame({
        'A': [1, 1, 2],
        'B': [1, 1, 2]
    })
    result = remove_duplicates(test_df)
    assert len(result) == 2
    assert not result.duplicated().any()

# Data quality assertions
def assert_data_quality(df):
    # No duplicates
    assert not df.duplicated().any(), "Data contains duplicates"

    # No missing values in critical columns
    critical_columns = ['id', 'date', 'amount']
    for col in critical_columns:
        if col in df.columns:
            assert not df[col].isnull().any(), f"Critical column {col} has missing valu
es"

    # Valid ranges
    if 'age' in df.columns:
        assert df['age'].between(0, 120).all(), "Invalid age values found"

    print("All data quality checks passed!")
```

```
# Run quality checks
assert_data_quality(df_clean)
```