rules\_dictionary.yaml

validations:

  - rule\_id: "R001"

    template\_name: "CustomerData"

    column\_name: "customer\_id"

    rule\_type: "Mandatory"

  - rule\_id: "R002"

    template\_name: "CustomerData"

    column\_name: "age"

    rule\_type: "Range"

  - rule\_id: "R003"

    template\_name: "CustomerData"

    column\_name: "email"

    rule\_type: "Pattern"

template:

  - feed\_name: "CustomerData"

    column\_name: "customer\_id"

    definition: "Unique identifier for customer"

    type\_name: "string"

    mandatory: true

    unique: true

  - feed\_name: "CustomerData"

    column\_name: "age"

    definition: "Customer age"

    type\_name: "integer"

    range:

      bottom: 0

      top: 120

      scope: "inclusive"

    mandatory: true

enumerations:

  country\_codes:

    - "USA"

    - "UK"

    - "IND"

    - "CHN"

    - "LKA"

pattern:

  email\_pattern: "^[a-zA-Z0-9.\_%+-]+@[a-zA-Z0-9.-]+\\.[a-zA-Z]{2,}$"

  phone\_pattern: "^\\+?[1-9][0-9]{7,14}$"

business\_rules:

  - rule\_id: "BR001"

    description: "Sum validation for specific countries"

    condition: "country in ('IND', 'CHN', 'LKA')"

    validation: "column1 + column2 >= column3"

    group\_by: ["sdsid", "extract\_month"]

reconciliation:

  - rec\_id: "REC001"

    source\_feed: "source\_table"

    target\_feed: "target\_table"

    dimensions: "column1;column2;column3"

    dimension\_values: "value1;(value2,value3);value4"

    measure\_field: "amount"

reporting:

  schedule: "daily"

  format:

    - "html"

    - "pdf"

    - "excel"

  recipients: ["team@company.com"]

  branding:

    logo\_path: "static/images/logo.png"

    company\_name: "Data Quality Framework"

    color\_scheme:

      primary: "#007bff"

      secondary: "#6c757d"

main.py

import yaml

import logging

from pathlib import Path

from datetime import datetime

from typing import Dict, List, Any

class DataQualityFramework:

    def \_\_init\_\_(self, config\_path: str = "../config/rules\_dictionary.yaml"):

        """Initialize the Data Quality Framework."""

        self.logger = self.\_setup\_logging()

        self.config = self.\_load\_config(config\_path)

        self.validation\_results = {}

    def \_setup\_logging(self) -> logging.Logger:

        """Set up logging configuration."""

        logger = logging.getLogger('dq\_framework')

        logger.setLevel(logging.INFO)

        # Create handlers

        c\_handler = logging.StreamHandler()

        f\_handler = logging.FileHandler('dq\_framework.log')

        # Create formatters and add it to handlers

        log\_format = '%(asctime)s - %(name)s - %(levelname)s - %(message)s'

        formatter = logging.Formatter(log\_format)

        c\_handler.setFormatter(formatter)

        f\_handler.setFormatter(formatter)

        # Add handlers to the logger

        logger.addHandler(c\_handler)

        logger.addHandler(f\_handler)

        return logger

    def \_load\_config(self, config\_path: str) -> Dict:

        """Load configuration from YAML file."""

        try:

            with open(config\_path, 'r') as file:

                return yaml.safe\_load(file)

        except Exception as e:

            self.logger.error(f"Error loading configuration: {str(e)}")

            raise

    def validate\_data(self, data\_source: Any) -> Dict:

        """Run all configured validations on the data source."""

        try:

            self.logger.info(f"Starting data validation at {datetime.now()}")

            # Run different types of validations

            self.\_run\_count\_validation(data\_source)

            self.\_run\_checksum\_validation(data\_source)

            self.\_run\_business\_rules(data\_source)

            self.\_run\_reconciliation(data\_source)

            self.\_run\_pattern\_checks(data\_source)

            self.\_run\_enumeration\_checks(data\_source)

            self.\_run\_mandatory\_checks(data\_source)

            self.\_run\_range\_checks(data\_source)

            self.\_run\_type\_checks(data\_source)

            self.\_run\_unique\_checks(data\_source)

            self.logger.info(f"Completed data validation at {datetime.now()}")

            return self.validation\_results

        except Exception as e:

            self.logger.error(f"Error during validation: {str(e)}")

            raise

    def \_run\_count\_validation(self, data\_source: Any) -> None:

        """Run count validation on the data source."""

        self.logger.info("Running count validation")

        # Implementation details here

        self.validation\_results['count\_validation'] = {

            'status': 'completed',

            'timestamp': datetime.now().isoformat()

        }

    def \_run\_checksum\_validation(self, data\_source: Any) -> None:

        """Run checksum validation on the data source."""

        self.logger.info("Running checksum validation")

        # Implementation details here

        self.validation\_results['checksum\_validation'] = {

            'status': 'completed',

            'timestamp': datetime.now().isoformat()

        }

    def \_run\_business\_rules(self, data\_source: Any) -> None:

        """Run business rule validations."""

        self.logger.info("Running business rule validations")

        # Implementation details here

        self.validation\_results['business\_rules'] = {

            'status': 'completed',

            'timestamp': datetime.now().isoformat()

        }

    def \_run\_reconciliation(self, data\_source: Any) -> None:

        """Run reconciliation checks."""

        self.logger.info("Running reconciliation checks")

        # Implementation details here

        self.validation\_results['reconciliation'] = {

            'status': 'completed',

            'timestamp': datetime.now().isoformat()

        }

    def \_run\_pattern\_checks(self, data\_source: Any) -> None:

        """Run pattern checks on the data."""

        self.logger.info("Running pattern checks")

        # Implementation details here

        self.validation\_results['pattern\_checks'] = {

            'status': 'completed',

            'timestamp': datetime.now().isoformat()

        }

    def \_run\_enumeration\_checks(self, data\_source: Any) -> None:

        """Run enumeration checks on the data."""

        self.logger.info("Running enumeration checks")

        # Implementation details here

        self.validation\_results['enumeration\_checks'] = {

            'status': 'completed',

            'timestamp': datetime.now().isoformat()

        }

    def \_run\_mandatory\_checks(self, data\_source: Any) -> None:

        """Run mandatory field checks."""

        self.logger.info("Running mandatory field checks")

        # Implementation details here

        self.validation\_results['mandatory\_checks'] = {

            'status': 'completed',

            'timestamp': datetime.now().isoformat()

        }

    def \_run\_range\_checks(self, data\_source: Any) -> None:

        """Run range checks on numeric fields."""

        self.logger.info("Running range checks")

        # Implementation details here

        self.validation\_results['range\_checks'] = {

            'status': 'completed',

            'timestamp': datetime.now().isoformat()

        }

    def \_run\_type\_checks(self, data\_source: Any) -> None:

        """Run data type checks."""

        self.logger.info("Running type checks")

        # Implementation details here

        self.validation\_results['type\_checks'] = {

            'status': 'completed',

            'timestamp': datetime.now().isoformat()

        }

    def \_run\_unique\_checks(self, data\_source: Any) -> None:

        """Run uniqueness checks."""

        self.logger.info("Running uniqueness checks")

        # Implementation details here

        self.validation\_results['unique\_checks'] = {

            'status': 'completed',

            'timestamp': datetime.now().isoformat()

        }

    def generate\_report(self) -> None:

        """Generate validation report."""

        try:

            self.logger.info("Generating validation report")

            # Report generation logic will be implemented in reporting module

            pass

        except Exception as e:

            self.logger.error(f"Error generating report: {str(e)}")

            raise

if \_\_name\_\_ == "\_\_main\_\_":

    # Example usage

    dq\_framework = DataQualityFramework()

    # Add implementation for data source connection and validation

report\_generator.py

import os

import json

import logging

from datetime import datetime

from typing import Dict, List, Optional

from pathlib import Path

import plotly.graph\_objects as go

import plotly.express as px

from jinja2 import Environment, FileSystemLoader

import pandas as pd

import weasyprint

class ReportGenerator:

    def \_\_init\_\_(self, config: Dict):

        """Initialize the Report Generator."""

        self.config = config

        self.logger = logging.getLogger('dq\_framework.reporting')

        self.template\_dir = Path(\_\_file\_\_).parent / 'templates'

        self.env = Environment(loader=FileSystemLoader(str(self.template\_dir)))

    def generate\_report(self, validation\_results: Dict, output\_format: List[str] = None) -> Dict[str, str]:

        """Generate validation report in specified formats."""

        if output\_format is None:

            output\_format = self.config['reporting']['format']

        report\_files = {}

        timestamp = datetime.now().strftime('%Y%m%d\_%H%M%S')

        try:

            # Generate report content

            report\_data = self.\_prepare\_report\_data(validation\_results)

            # Generate reports in specified formats

            for fmt in output\_format:

                if fmt.lower() == 'html':

                    report\_files['html'] = self.\_generate\_html\_report(report\_data, timestamp)

                elif fmt.lower() == 'pdf':

                    report\_files['pdf'] = self.\_generate\_pdf\_report(report\_data, timestamp)

                elif fmt.lower() == 'excel':

                    report\_files['excel'] = self.\_generate\_excel\_report(report\_data, timestamp)

            return report\_files

        except Exception as e:

            self.logger.error(f"Error generating report: {str(e)}")

            raise

    def \_prepare\_report\_data(self, validation\_results: Dict) -> Dict:

        """Prepare data for report generation."""

        return {

            'summary': self.\_generate\_summary(validation\_results),

            'detailed\_results': self.\_prepare\_detailed\_results(validation\_results),

            'visualizations': self.\_generate\_visualizations(validation\_results),

            'timestamp': datetime.now().isoformat(),

            'branding': self.config['reporting']['branding']

        }

    def \_generate\_summary(self, validation\_results: Dict) -> Dict:

        """Generate executive summary of validation results."""

        total\_validations = len(validation\_results)

        passed\_validations = sum(1 for result in validation\_results.values()

                               if result.get('status') == 'completed')

        return {

            'total\_validations': total\_validations,

            'passed\_validations': passed\_validations,

            'failed\_validations': total\_validations - passed\_validations,

            'success\_rate': (passed\_validations / total\_validations \* 100) if total\_validations > 0 else 0

        }

    def \_prepare\_detailed\_results(self, validation\_results: Dict) -> List[Dict]:

        """Prepare detailed validation results for reporting."""

        detailed\_results = []

        for validation\_type, result in validation\_results.items():

            detailed\_results.append({

                'validation\_type': validation\_type,

                'status': result.get('status'),

                'timestamp': result.get('timestamp'),

                'details': result.get('details', {}),

                'error\_samples': self.\_get\_error\_samples(result)

            })

        return detailed\_results

    def \_generate\_visualizations(self, validation\_results: Dict) -> Dict:

        """Generate visualization data for the report."""

        # Create validation status pie chart

        status\_counts = {'Passed': 0, 'Failed': 0}

        for result in validation\_results.values():

            if result.get('status') == 'completed':

                status\_counts['Passed'] += 1

            else:

                status\_counts['Failed'] += 1

        status\_pie = go.Figure(data=[go.Pie(

            labels=list(status\_counts.keys()),

            values=list(status\_counts.values()),

            marker\_colors=['#28a745', '#dc3545']

        )])

        # Create validation type bar chart

        validation\_types = {}

        for v\_type, result in validation\_results.items():

            validation\_types[v\_type] = len(result.get('details', {}))

        type\_bar = go.Figure(data=[go.Bar(

            x=list(validation\_types.keys()),

            y=list(validation\_types.values())

        )])

        return {

            'status\_pie': status\_pie.to\_html(full\_html=False),

            'type\_bar': type\_bar.to\_html(full\_html=False)

        }

    def \_generate\_html\_report(self, report\_data: Dict, timestamp: str) -> str:

        """Generate HTML report."""

        template = self.env.get\_template('report\_template.html')

        output\_path = f'reports/validation\_report\_{timestamp}.html'

        # Ensure reports directory exists

        os.makedirs('reports', exist\_ok=True)

        # Generate HTML report

        html\_content = template.render(\*\*report\_data)

        with open(output\_path, 'w') as f:

            f.write(html\_content)

        return output\_path

    def \_generate\_pdf\_report(self, report\_data: Dict, timestamp: str) -> str:

        """Generate PDF report."""

        html\_path = self.\_generate\_html\_report(report\_data, timestamp)

        pdf\_path = f'reports/validation\_report\_{timestamp}.pdf'

        # Convert HTML to PDF

        weasyprint.HTML(filename=html\_path).write\_pdf(pdf\_path)

        return pdf\_path

    def \_generate\_excel\_report(self, report\_data: Dict, timestamp: str) -> str:

        """Generate Excel report with multiple sheets."""

        excel\_path = f'reports/validation\_report\_{timestamp}.xlsx'

        # Create Excel writer

        with pd.ExcelWriter(excel\_path, engine='openpyxl') as writer:

            # Summary sheet

            summary\_df = pd.DataFrame([report\_data['summary']])

            summary\_df.to\_excel(writer, sheet\_name='Summary', index=False)

            # Detailed results sheet

            detailed\_df = pd.DataFrame(report\_data['detailed\_results'])

            detailed\_df.to\_excel(writer, sheet\_name='Detailed Results', index=False)

        return excel\_path

    def \_get\_error\_samples(self, result: Dict, max\_samples: int = 5) -> List[Dict]:

        """Get sample of error records from validation result."""

        error\_samples = result.get('error\_samples', [])

        return error\_samples[:max\_samples]

    def email\_report(self, report\_files: Dict[str, str]) -> None:

        """Email the generated reports to configured recipients."""

        # Email functionality to be implemented

        pass

    def \_create\_report\_directory(self) -> None:

        """Create reports directory if it doesn't exist."""

        os.makedirs('reports', exist\_ok=True)

report\_template.html

<!DOCTYPE html>

<html lang="en">

<head>

    <meta charset="UTF-8">

    <meta name="viewport" content="width=device-width, initial-scale=1.0">

    <title>Data Quality Validation Report</title>

    <!-- Google Fonts -->

    <link href="https://fonts.googleapis.com/css2?family=Inter:wght@400;500;600;700&display=swap" rel="stylesheet">

    <!-- Tailwind CSS -->

    <script src="https://cdn.tailwindcss.com"></script>

    <style>

        body {

            font-family: 'Inter', sans-serif;

            background-color: #f8fafc;

        }

        .card {

            background-color: white;

            border-radius: 0.5rem;

            box-shadow: 0 1px 3px 0 rgb(0 0 0 / 0.1);

            padding: 1.5rem;

            margin-bottom: 1.5rem;

        }

        .status-badge {

            padding: 0.25rem 0.75rem;

            border-radius: 9999px;

            font-weight: 500;

            font-size: 0.875rem;

        }

        .status-badge.success {

            background-color: #dcfce7;

            color: #166534;

        }

        .status-badge.error {

            background-color: #fee2e2;

            color: #991b1b;

        }

        .metric-card {

            background-color: white;

            border-radius: 0.5rem;

            padding: 1rem;

            text-align: center;

        }

        .metric-value {

            font-size: 2rem;

            font-weight: 600;

            color: #1e293b;

        }

        .metric-label {

            font-size: 0.875rem;

            color: #64748b;

            margin-top: 0.25rem;

        }

        table {

            width: 100%;

            border-collapse: collapse;

        }

        th, td {

            padding: 0.75rem 1rem;

            text-align: left;

            border-bottom: 1px solid #e2e8f0;

        }

        th {

            background-color: #f8fafc;

            font-weight: 600;

            color: #475569;

        }

        .chart-container {

            margin-top: 1.5rem;

            padding: 1rem;

            background-color: white;

            border-radius: 0.5rem;

            box-shadow: 0 1px 3px 0 rgb(0 0 0 / 0.1);

        }

    </style>

</head>

<body class="p-8">

    <!-- Header -->

    <header class="mb-8">

        <div class="flex items-center justify-between">

            <div>

                <h1 class="text-3xl font-bold text-gray-900">Data Quality Validation Report</h1>

                <p class="text-gray-500 mt-2">Generated on {{ timestamp }}</p>

            </div>

            {% if branding.logo\_path %}

            <img src="{{ branding.logo\_path }}" alt="{{ branding.company\_name }}" class="h-12">

            {% endif %}

        </div>

    </header>

    <!-- Executive Summary -->

    <section class="mb-8">

        <h2 class="text-2xl font-semibold text-gray-900 mb-4">Executive Summary</h2>

        <div class="grid grid-cols-1 md:grid-cols-4 gap-4">

            <div class="metric-card">

                <div class="metric-value">{{ summary.total\_validations }}</div>

                <div class="metric-label">Total Validations</div>

            </div>

            <div class="metric-card">

                <div class="metric-value text-green-600">{{ summary.passed\_validations }}</div>

                <div class="metric-label">Passed</div>

            </div>

            <div class="metric-card">

                <div class="metric-value text-red-600">{{ summary.failed\_validations }}</div>

                <div class="metric-label">Failed</div>

            </div>

            <div class="metric-card">

                <div class="metric-value text-blue-600">{{ "%.2f"|format(summary.success\_rate) }}%</div>

                <div class="metric-label">Success Rate</div>

            </div>

        </div>

    </section>

    <!-- Visualizations -->

    <section class="mb-8">

        <h2 class="text-2xl font-semibold text-gray-900 mb-4">Validation Results Overview</h2>

        <div class="grid grid-cols-1 md:grid-cols-2 gap-4">

            <div class="chart-container">

                <h3 class="text-lg font-medium text-gray-900 mb-4">Status Distribution</h3>

                {{ visualizations.status\_pie | safe }}

            </div>

            <div class="chart-container">

                <h3 class="text-lg font-medium text-gray-900 mb-4">Validation Types</h3>

                {{ visualizations.type\_bar | safe }}

            </div>

        </div>

    </section>

    <!-- Detailed Results -->

    <section>

        <h2 class="text-2xl font-semibold text-gray-900 mb-4">Detailed Results</h2>

        <div class="card">

            <table>

                <thead>

                    <tr>

                        <th>Validation Type</th>

                        <th>Status</th>

                        <th>Timestamp</th>

                        <th>Details</th>

                    </tr>

                </thead>

                <tbody>

                    {% for result in detailed\_results %}

                    <tr>

                        <td class="font-medium">{{ result.validation\_type }}</td>

                        <td>

                            <span class="status-badge {{ 'success' if result.status == 'completed' else 'error' }}">

                                {{ result.status }}

                            </span>

                        </td>

                        <td>{{ result.timestamp }}</td>

                        <td>

                            {% if result.error\_samples %}

                            <details>

                                <summary class="cursor-pointer text-blue-600">View Details</summary>

                                <div class="mt-2">

                                    <h4 class="font-medium mb-2">Error Samples:</h4>

                                    <ul class="list-disc pl-5">

                                    {% for error in result.error\_samples %}

                                        <li>{{ error }}</li>

                                    {% endfor %}

                                    </ul>

                                </div>

                            </details>

                            {% endif %}

                        </td>

                    </tr>

                    {% endfor %}

                </tbody>

            </table>

        </div>

    </section>

    <!-- Footer -->

    <footer class="mt-8 text-center text-gray-500 text-sm">

        <p>Generated by Data Quality Validation Framework</p>

        {% if branding.company\_name %}

        <p class="mt-1">{{ branding.company\_name }}</p>

        {% endif %}

    </footer>

</body>

</html>

validations.py

import logging

from typing import Dict, List, Any, Optional, Tuple

from datetime import datetime

import pandas as pd

import great\_expectations as ge

from great\_expectations.dataset import PandasDataset

class DataValidator:

    def \_\_init\_\_(self, config: Dict):

        """Initialize the Data Validator."""

        self.config = config

        self.logger = logging.getLogger('dq\_framework.validations')

        self.validation\_results = {}

    def validate\_dataset(self, data: pd.DataFrame, source\_name: str) -> Dict:

        """Run all configured validations on the dataset."""

        try:

            # Convert to Great Expectations dataset

            ge\_dataset = ge.from\_pandas(data)

            self.validation\_results = {

                'source\_name': source\_name,

                'timestamp': datetime.now().isoformat(),

                'validations': {}

            }

            # Run all configured validations

            self.\_run\_count\_validation(ge\_dataset)

            self.\_run\_checksum\_validation(ge\_dataset)

            self.\_run\_duplicate\_check(ge\_dataset)

            self.\_run\_pattern\_check(ge\_dataset)

            self.\_run\_enumeration\_check(ge\_dataset)

            self.\_run\_mandatory\_check(ge\_dataset)

            self.\_run\_range\_check(ge\_dataset)

            self.\_run\_type\_check(ge\_dataset)

            self.\_run\_unique\_check(ge\_dataset)

            self.\_run\_business\_rules(ge\_dataset)

            return self.validation\_results

        except Exception as e:

            self.logger.error(f"Error validating dataset: {str(e)}")

            raise

    def \_run\_count\_validation(self, dataset: PandasDataset) -> None:

        """Validate record count against expected value."""

        validation\_type = 'count\_validation'

        self.logger.info(f"Running {validation\_type}")

        try:

            # Get expected count from configuration

            expected\_count = self.config.get('expected\_count', 0)

            # Validate count

            actual\_count = len(dataset)

            result = {

                'status': 'completed' if actual\_count == expected\_count else 'failed',

                'details': {

                    'expected\_count': expected\_count,

                    'actual\_count': actual\_count,

                    'difference': actual\_count - expected\_count

                }

            }

            self.validation\_results['validations'][validation\_type] = result

        except Exception as e:

            self.logger.error(f"Error in {validation\_type}: {str(e)}")

            self.\_log\_validation\_error(validation\_type, str(e))

    def \_run\_checksum\_validation(self, dataset: PandasDataset) -> None:

        """Validate checksum of specified columns."""

        validation\_type = 'checksum\_validation'

        self.logger.info(f"Running {validation\_type}")

        try:

            checksum\_columns = self.config.get('checksum\_columns', [])

            checksums = {}

            for column in checksum\_columns:

                if column in dataset.columns:

                    checksums[column] = dataset[column].sum()

            self.validation\_results['validations'][validation\_type] = {

                'status': 'completed',

                'details': {'checksums': checksums}

            }

        except Exception as e:

            self.logger.error(f"Error in {validation\_type}: {str(e)}")

            self.\_log\_validation\_error(validation\_type, str(e))

    def \_run\_duplicate\_check(self, dataset: PandasDataset) -> None:

        """Check for duplicate records."""

        validation\_type = 'duplicate\_check'

        self.logger.info(f"Running {validation\_type}")

        try:

            # Get columns to check for duplicates

            check\_columns = self.config.get('duplicate\_check\_columns', dataset.columns.tolist())

            # Find duplicates

            duplicates = dataset.duplicated(subset=check\_columns, keep='first')

            duplicate\_count = duplicates.sum()

            # Get sample of duplicate records

            duplicate\_samples = dataset[duplicates].head(5).to\_dict('records')

            result = {

                'status': 'completed' if duplicate\_count == 0 else 'failed',

                'details': {

                    'duplicate\_count': int(duplicate\_count),

                    'checked\_columns': check\_columns,

                    'duplicate\_samples': duplicate\_samples

                }

            }

            self.validation\_results['validations'][validation\_type] = result

        except Exception as e:

            self.logger.error(f"Error in {validation\_type}: {str(e)}")

            self.\_log\_validation\_error(validation\_type, str(e))

    def \_run\_pattern\_check(self, dataset: PandasDataset) -> None:

        """Check if values match specified patterns."""

        validation\_type = 'pattern\_check'

        self.logger.info(f"Running {validation\_type}")

        try:

            pattern\_rules = self.config.get('pattern', {})

            results = {}

            for column, pattern in pattern\_rules.items():

                if column in dataset.columns:

                    # Use Great Expectations to check pattern

                    validation = dataset.expect\_column\_values\_to\_match\_regex(

                        column,

                        pattern

                    )

                    results[column] = {

                        'success': validation.success,

                        'unexpected\_count': validation.result['unexpected\_count'],

                        'unexpected\_samples': validation.result['unexpected\_list'][:5]

                    }

            self.validation\_results['validations'][validation\_type] = {

                'status': 'completed' if all(r['success'] for r in results.values()) else 'failed',

                'details': results

            }

        except Exception as e:

            self.logger.error(f"Error in {validation\_type}: {str(e)}")

            self.\_log\_validation\_error(validation\_type, str(e))

    def \_run\_enumeration\_check(self, dataset: PandasDataset) -> None:

        """Check if values are within allowed enumerations."""

        validation\_type = 'enumeration\_check'

        self.logger.info(f"Running {validation\_type}")

        try:

            enum\_rules = self.config.get('enumerations', {})

            results = {}

            for column, allowed\_values in enum\_rules.items():

                if column in dataset.columns:

                    validation = dataset.expect\_column\_values\_to\_be\_in\_set(

                        column,

                        allowed\_values

                    )

                    results[column] = {

                        'success': validation.success,

                        'unexpected\_count': validation.result['unexpected\_count'],

                        'unexpected\_samples': validation.result['unexpected\_list'][:5]

                    }

            self.validation\_results['validations'][validation\_type] = {

                'status': 'completed' if all(r['success'] for r in results.values()) else 'failed',

                'details': results

            }

        except Exception as e:

            self.logger.error(f"Error in {validation\_type}: {str(e)}")

            self.\_log\_validation\_error(validation\_type, str(e))

    def \_run\_mandatory\_check(self, dataset: PandasDataset) -> None:

        """Check for null values in mandatory fields."""

        validation\_type = 'mandatory\_check'

        self.logger.info(f"Running {validation\_type}")

        try:

            mandatory\_fields = [

                field['column\_name']

                for field in self.config.get('template', [])

                if field.get('mandatory', False)

            ]

            results = {}

            for field in mandatory\_fields:

                if field in dataset.columns:

                    validation = dataset.expect\_column\_values\_to\_not\_be\_null(field)

                    results[field] = {

                        'success': validation.success,

                        'null\_count': validation.result['unexpected\_count'],

                        'null\_examples': validation.result['unexpected\_list'][:5]

                    }

            self.validation\_results['validations'][validation\_type] = {

                'status': 'completed' if all(r['success'] for r in results.values()) else 'failed',

                'details': results

            }

        except Exception as e:

            self.logger.error(f"Error in {validation\_type}: {str(e)}")

            self.\_log\_validation\_error(validation\_type, str(e))

    def \_run\_range\_check(self, dataset: PandasDataset) -> None:

        """Check if numeric values are within specified ranges."""

        validation\_type = 'range\_check'

        self.logger.info(f"Running {validation\_type}")

        try:

            range\_rules = [

                field for field in self.config.get('template', [])

                if 'range' in field

            ]

            results = {}

            for rule in range\_rules:

                column = rule['column\_name']

                if column in dataset.columns:

                    range\_config = rule['range']

                    validation = dataset.expect\_column\_values\_to\_be\_between(

                        column,

                        min\_value=range\_config['bottom'],

                        max\_value=range\_config['top'],

                        include\_min=range\_config.get('scope') == 'inclusive',

                        include\_max=range\_config.get('scope') == 'inclusive'

                    )

                    results[column] = {

                        'success': validation.success,

                        'unexpected\_count': validation.result['unexpected\_count'],

                        'unexpected\_samples': validation.result['unexpected\_list'][:5]

                    }

            self.validation\_results['validations'][validation\_type] = {

                'status': 'completed' if all(r['success'] for r in results.values()) else 'failed',

                'details': results

            }

        except Exception as e:

            self.logger.error(f"Error in {validation\_type}: {str(e)}")

            self.\_log\_validation\_error(validation\_type, str(e))

    def \_run\_type\_check(self, dataset: PandasDataset) -> None:

        """Check if columns have the correct data type."""

        validation\_type = 'type\_check'

        self.logger.info(f"Running {validation\_type}")

        try:

            type\_rules = [

                field for field in self.config.get('template', [])

                if 'type\_name' in field

            ]

            results = {}

            for rule in type\_rules:

                column = rule['column\_name']

                if column in dataset.columns:

                    expected\_type = rule['type\_name']

                    # Map type names to pandas dtypes

                    type\_mapping = {

                        'string': 'object',

                        'integer': 'int64',

                        'float': 'float64',

                        'boolean': 'bool',

                        'date': 'datetime64[ns]'

                    }

                    actual\_type = str(dataset[column].dtype)

                    expected\_pandas\_type = type\_mapping.get(expected\_type.lower(), expected\_type)

                    results[column] = {

                        'success': actual\_type == expected\_pandas\_type,

                        'expected\_type': expected\_pandas\_type,

                        'actual\_type': actual\_type

                    }

            self.validation\_results['validations'][validation\_type] = {

                'status': 'completed' if all(r['success'] for r in results.values()) else 'failed',

                'details': results

            }

        except Exception as e:

            self.logger.error(f"Error in {validation\_type}: {str(e)}")

            self.\_log\_validation\_error(validation\_type, str(e))

    def \_run\_unique\_check(self, dataset: PandasDataset) -> None:

        """Check if specified columns have unique values."""

        validation\_type = 'unique\_check'

        self.logger.info(f"Running {validation\_type}")

        try:

            unique\_fields = [

                field['column\_name']

                for field in self.config.get('template', [])

                if field.get('unique', False)

            ]

            results = {}

            for field in unique\_fields:

                if field in dataset.columns:

                    validation = dataset.expect\_column\_values\_to\_be\_unique(field)

                    results[field] = {

                        'success': validation.success,

                        'duplicate\_count': validation.result['unexpected\_count'],

                        'duplicate\_examples': validation.result['unexpected\_list'][:5]

                    }

            self.validation\_results['validations'][validation\_type] = {

                'status': 'completed' if all(r['success'] for r in results.values()) else 'failed',

                'details': results

            }

        except Exception as e:

            self.logger.error(f"Error in {validation\_type}: {str(e)}")

            self.\_log\_validation\_error(validation\_type, str(e))

    def \_run\_business\_rules(self, dataset: PandasDataset) -> None:

        """Run custom business rule validations."""

        validation\_type = 'business\_rules'

        self.logger.info(f"Running {validation\_type}")

        try:

            business\_rules = self.config.get('business\_rules', [])

            results = {}

            for rule in business\_rules:

                rule\_id = rule['rule\_id']

                # Create a query string for the business rule

                query = f"({rule['condition']}) & ({rule['validation']})"

                # Apply the rule and get violations

                try:

                    mask = dataset.query(query)

                    violations = dataset[~mask]

                    results[rule\_id] = {

                        'success': len(violations) == 0,

                        'violation\_count': len(violations),

                        'violation\_samples': violations.head(5).to\_dict('records')

                    }

                except Exception as rule\_error:

                    results[rule\_id] = {

                        'success': False,

                        'error': str(rule\_error)

                    }

            self.validation\_results['validations'][validation\_type] = {

                'status': 'completed' if all(r['success'] for r in results.values()) else 'failed',

                'details': results

            }

        except Exception as e:

            self.logger.error(f"Error in {validation\_type}: {str(e)}")

            self.\_log\_validation\_error(validation\_type, str(e))

    def \_log\_validation\_error(self, validation\_type: str, error\_message: str) -> None:

        """Log validation error and update validation results."""

        self.validation\_results['validations'][validation\_type] = {

            'status': 'error',

            'error': error\_message

        }

example\_usage.py

import pandas as pd

from dq\_framework.main import DataQualityFramework

from pathlib import Path

def load\_sample\_data() -> pd.DataFrame:

    """Create a sample dataset for demonstration."""

    return pd.DataFrame({

        'customer\_id': ['C001', 'C002', 'C003', 'C002', 'C005'],  # Contains duplicate

        'age': [25, 150, -5, 35, 40],  # Contains out of range values

        'email': [

            'valid@email.com',

            'invalid.email',  # Invalid pattern

            'another@email.com',

            'test@test.com',

            None  # Missing value

        ],

        'country': ['USA', 'INVALID', 'IND', 'CHN', 'LKA'],  # Contains invalid enum

        'column1': [100, 200, 300, 400, 500],

        'column2': [50, 100, 150, 200, 250],

        'column3': [200, 250, 400, 550, 700]  # Business rule violation

    })

def main():

    """Example usage of the Data Quality Validation Framework."""

    try:

        # Initialize the framework

        dq\_framework = DataQualityFramework()

        # Load sample data

        print("Loading sample data...")

        data = load\_sample\_data()

        # Run validations

        print("\nRunning validations...")

        validation\_results = dq\_framework.validate\_data(data)

        # Generate report

        print("\nGenerating validation report...")

        dq\_framework.generate\_report()

        print("\nValidation process completed successfully!")

        print("Check the 'reports' directory for the detailed validation report.")

    except Exception as e:

        print(f"\nError: {str(e)}")

        raise

if \_\_name\_\_ == "\_\_main\_\_":

    main()

README.md

**# Data Quality Validation Framework**

A comprehensive data quality validation framework built using Great Expectations that performs various validation checks on different data sources and generates detailed, beautiful reports.

**## Features**

**### Validation Types**

1. Count Validation

2. Checksum Validation

3. Business Rule Validation

4. Reconciliation Logic

5. Duplicate Check

6. Pattern Check

7. Enumeration Check

8. Mandatory Check

9. Range Check

10. Type Check

11. Unique Check

**### Supported Data Sources**

- SQL Server Tables

- Feed Files

- Flat Files

- Parquet Files

- AWS S3 Bucket Feed Files

**### Report Generation**

- Interactive HTML Reports

- PDF Reports

- Excel Reports with Multiple Sheets

- Email Distribution Capability

**## Installation**

1. Clone the repository:

```bash

git clone <repository-url>

cd data-quality-framework

```

2. Install required dependencies:

```bash

pip install -r requirements.txt

```

**## Configuration**

The framework uses a YAML configuration file (`config/rules\_dictionary.yaml`) that defines:

1. **\*\*Validations\*\***: List of unique rules defining which type of rules need to be run

   - Rule\_ID

   - Template Name

   - Columns Name

   - RuleType (Mandatory, Range, Type, Unique, etc.)

2. **\*\*Template\*\***: Description of templates with columns:

   - Feed Name

   - Column Name

   - Definition

   - Enumeration

   - Type\_Name

   - Range (Bottom, Top, Scope)

   - Mandatory

   - Unique

3. **\*\*Enumerations\*\***: List of expected values for non-numeric fields

4. **\*\*Pattern\*\***: Pattern expectations for specific feeds

5. **\*\*Business Rules\*\***: Custom business validations

6. **\*\*Reconciliations\*\***: Source/Target reconciliation rules

**## Usage**

**### Basic Usage**

```python

from dq\_framework.main import DataQualityFramework

# Initialize the framework

dq\_framework = DataQualityFramework()

# Load your data (example with pandas DataFrame)

data = pd.read\_csv('your\_data.csv')

# Run validations

validation\_results = dq\_framework.validate\_data(data)

# Generate report

dq\_framework.generate\_report()

```

**### Example Script**

An example script (`example\_usage.py`) is provided that demonstrates the framework's usage with sample data:

```bash

python example\_usage.py

```

**## Report Types**

**### 1. HTML Report**

- Interactive visualizations

- Detailed validation results

- Expandable error samples

- Modern, responsive design

- Filtering and sorting capabilities

**### 2. PDF Report**

- Professional formatting

- Printer-friendly

- Ideal for sharing and archiving

**### 3. Excel Report**

- Multiple worksheets for different aspects

- Detailed error logs

- Pivot-table ready format

**## Report Sections**

1. **\*\*Executive Summary\*\***

   - Overall validation status

   - Key metrics

   - Success rate

2. **\*\*Validation Details\*\***

   - Results for each validation type

   - Error samples

   - Detailed statistics

3. **\*\*Visualizations\*\***

   - Status distribution charts

   - Error trend analysis

   - Type-wise validation results

**## Customization**

**### Adding New Validation Types**

1. Add the validation configuration in `rules\_dictionary.yaml`

2. Implement the validation logic in `dq\_framework/validations.py`

3. Update the report template to display the new validation results

**### Customizing Reports**

1. Modify the HTML template in `dq\_framework/reporting/templates/report\_template.html`

2. Update the CSS styling

3. Add new visualizations in `report\_generator.py`

**## Best Practices**

1. **\*\*Configuration Management\*\***

   - Keep sensitive information in environment variables

   - Use separate configurations for different environments

2. **\*\*Data Source Handling\*\***

   - Implement proper error handling for data source connections

   - Use connection pooling for database connections

3. **\*\*Report Distribution\*\***

   - Set up automated report distribution for critical validations

   - Implement proper access controls for reports

**## Error Handling**

The framework implements comprehensive error handling:

- Detailed error logging

- Error samples in reports

- Graceful failure handling

- Error notifications

**## Contributing**

1. Fork the repository

2. Create a feature branch

3. Commit your changes

4. Push to the branch

5. Create a Pull Request

**## License**

This project is licensed under the MIT License - see the LICENSE file for details.



