Pandas in Python

Working with Files

Course Overview

01

Date Fundamentals

Master pandas datetime objects, parsing strategies, and timezone handling for robust temporal data analysis

02

File Format Mastery

Work confidently with CSV, Excel, JSON, and Parquet files using pandas I/O operations

03

Enterprise Integration

Connect to Snowflake and SAP HANA databases for seamless data pipeline development

04

Best Practices

Implement performance optimization techniques and error handling strategies

Date Handling Fundamentals

Dates are among the most critical data types in analytics. Pandas provides powerful tools for parsing, manipulating, and analyzing temporal data efficiently.

Understanding Pandas Datetime Objects

Key Concepts

- `datetime64[ns]` - Native pandas datetime type
- `Timestamp` - Scalar datetime representation
- `DatetimeIndex` - Time-based index for DataFrames
- `Timedelta` - Duration between datetime objects

These objects provide nanosecond precision and integrate seamlessly with pandas operations for high-performance temporal analysis.

Basic Date Creation

```
import pandas as pd
from datetime import datetime

# Create timestamps
ts = pd.Timestamp('2024-01-15')
ts_now = pd.Timestamp.now()

# Create date range
dates = pd.date_range('2024-01-01',
 periods=10,
 freq='D')
print(dates)
```

Parsing Dates from Strings

to_datetime() Method

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

# Custom format
df['date'] = pd.to_datetime(df['date_str'],
                            format='%Y-%m-%d')

# Handle errors gracefully
df['date'] = pd.to_datetime(df['date_str'],
                            errors='coerce')
```

Read Functions Integration

```
# Parse during file reading
df = pd.read_csv('data.csv',
 parse_dates=['date_column'])

# Multiple date columns
df = pd.read_csv('data.csv',
 parse_dates=['start_date', 'end_date'])

# Custom date parser
dateparse = lambda x: pd.datetime.strptime(x, '%Y-%m-%d')
df = pd.read_csv('data.csv',
 date_parser=dateparse)
```

Advanced Date Manipulation

1

Extract Components

```
df['year'] = df['date'].dt.year
df['month'] = df['date'].dt.month
df['weekday'] = df['date'].dt.dayname()
df['quarter'] = df['date'].dt.quarter
```

2

Date Arithmetic

```
# Add/subtract time periods
df['future'] = df['date'] +
pd.Timedelta(days=30)
df['past'] = df['date'] -
pd.DateOffset(months=1)

# Calculate differences
df['days_diff'] = (df['end_date'] -
df['start_date']).dt.days
```

3

Resampling

```
# Group by time periods
monthly =
df.set_index('date').resample('M').sum()
weekly =
df.set_index('date').resample('W').mean(
)

# Custom aggregations
daily_stats =
df.groupby(df['date'].dt.date).agg({
 'value': ['mean', 'max', 'count']
})
```

Timezone Handling

Working with Timezones

Enterprise data often comes from multiple timezones. Pandas provides robust timezone support for consistent temporal analysis across global datasets.

- Localize naive timestamps to specific timezones
- Convert between different timezone representations
- Handle daylight saving time transitions automatically
- Maintain timezone awareness in calculations

Timezone Operations

```
# Localize to timezone
df['date_tz'] = df['date'].dt.tz_localize('UTC')

# Convert timezone
df['date_est'] = df['date_tz'].dt.tz_convert('US/Eastern')

# Create timezone-aware dates
dates_utc = pd.date_range('2024-01-01',
 periods=5,
 tz='UTC')

# Remove timezone info
df['date_naive'] = df['date_tz'].dt.tz_localize(None)
```

File Format Mastery

Modern data engineering requires fluency across multiple file formats. Master the most common formats for seamless data integration.

CSV Files - The Universal Format

Reading CSV Files

```
# Basic read
df = pd.read_csv('data.csv')

# Common parameters
df = pd.read_csv('data.csv',
                 sep=',',          # Delimiter
                 header=0,         # Header row
                 index_col=0,      # Index
column
                 usecols=['A', 'B'], # Select
columns
                 dtype={'col': str}, # Data
types
                 na_values=['N/A'],  # Missing
values
                 encoding='utf-8')   # Text
encoding
```

Writing CSV Files

```
# Basic write
df.to_csv('output.csv')

# Advanced options
df.to_csv('output.csv',
          index=False,       # Exclude index
          sep='|',           # Custom delimiter
          na_rep='NULL',     # Missing value
representation
          float_format='%.2f', # Number
formatting
          date_format='%Y-%m-%d', #
Date formatting
          encoding='utf-8')   # Text
encoding
```

Performance Tips

- Use `chunksize` for large files
- Specify `dtype` to avoid inference
- Use `usecols` to read only needed columns
- Consider `low_memory=False` for mixed types

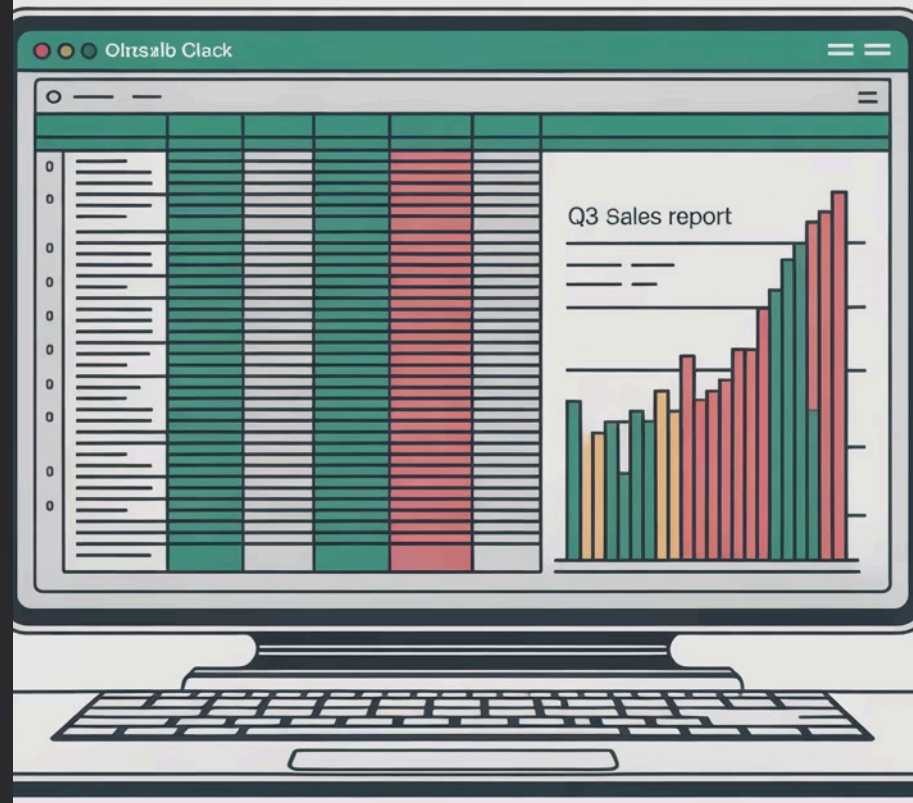
Excel Files - Enterprise Standard

Reading Excel Files

```
# Single sheet
df = pd.read_excel('data.xlsx',
sheet_name='Sheet1')

# Multiple sheets
dfs = pd.read_excel('data.xlsx',
sheet_name=None)

# Specific range
df = pd.read_excel('data.xlsx',
                   sheet_name='Data',
                   usecols='A:E',
                   skiprows=2,
                   nrows=100)

# With date parsing
df = pd.read_excel('data.xlsx',
                   parse_dates=['date_col'])
```

Writing Excel Files

```
# Single sheet
df.to_excel('output.xlsx',
sheet_name='Data')

# Multiple sheets
with pd.ExcelWriter('output.xlsx') as
writer:
 df1.to_excel(writer,
sheet_name='Sheet1')
 df2.to_excel(writer,
sheet_name='Sheet2')

# Formatting options
df.to_excel('output.xlsx',
 index=False,
 startrow=1,
 startcol=1,
 freeze_panes=(1,0))
```



JSON and Parquet - Modern Formats

JSON Operations

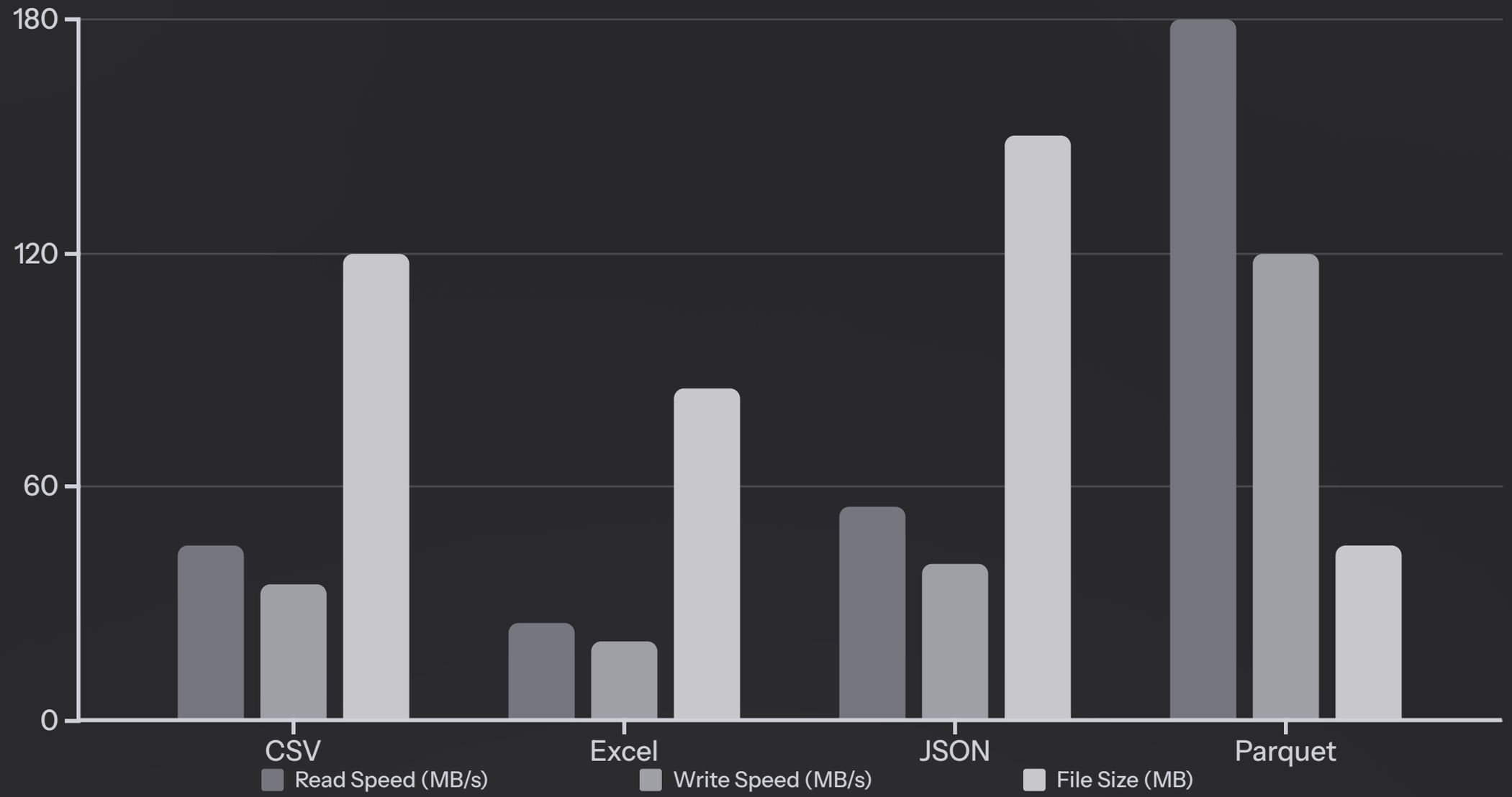
```
# Read JSON
df = pd.read_json('data.json')

# Nested JSON
df = pd.json_normalize(json_data)

# Write JSON
df.to_json('output.json',
          orient='records',
          indent=2)

# Handle nested structures
df = pd.json_normalize(data,
                       record_path=['items'],
                       meta=['id', 'name'])
```

Parquet Benefits

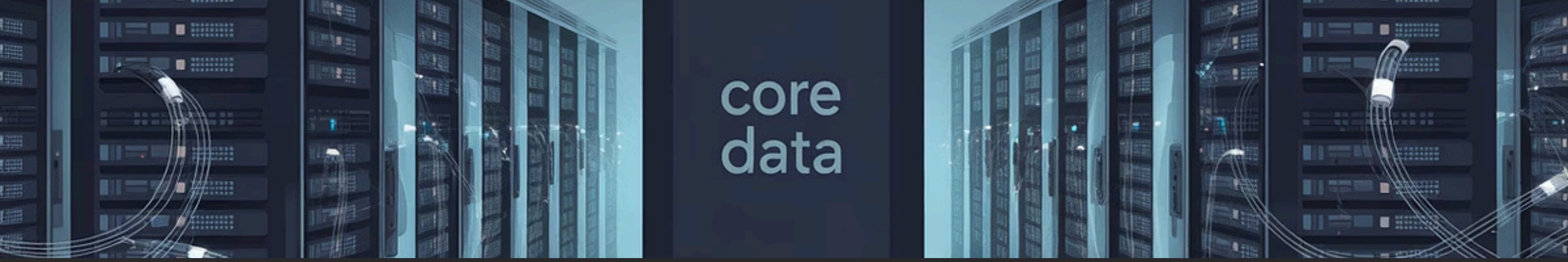
- Columnar storage format
- Excellent compression ratios
- Fast read/write operations
- Schema preservation
- Cross-language compatibility

```
# Read/write Parquet
df = pd.read_parquet('data.parquet')
df.to_parquet('output.parquet',
 compression='snappy')
```

File I/O Performance Comparison



Parquet significantly outperforms traditional formats in both speed and storage efficiency, making it ideal for large-scale data processing pipelines.



Enterprise Database Integration

Connect pandas to enterprise data platforms for seamless analytics workflows. Master Snowflake and SAP HANA integration patterns.

Connecting to Snowflake

Install Dependencies

```
pip install snowflake-connector-
python
pip install snowflake-sqlalchemy
```

Essential packages for Snowflake connectivity and SQLAlchemy integration with pandas.

Establish Connection

```
import snowflake.connector
from sqlalchemy import create_engine

# Connection parameters
conn_params = {
    'user': 'your_username',
    'password': 'your_password',
    'account': 'your_account',
    'warehouse': 'COMPUTE_WH',
    'database': 'YOUR_DB',
    'schema': 'PUBLIC'
}

# Create engine
engine =
create_engine(URL(**conn_params))
```

Execute Queries

```
# Read data
query = """
SELECT * FROM sales_data
WHERE date >= '2024-01-01'
ORDER BY date DESC
"""

df = pd.read_sql(query, engine)

# Write data
df.to_sql('new_table', engine,
 if_exists='replace',
 index=False)
```

Snowflake Best Practices

Performance Optimization

- Use warehouse size appropriate for workload
- Implement result caching strategies
- Leverage clustering keys for large tables
- Use COPY commands for bulk data loads
- Optimize query predicates and joins

Pro Tip: Use Snowflake's query profiler to identify performance bottlenecks and optimize accordingly.

Security and Authentication

```
# Using key pair authentication
from cryptography.hazmat.primitives import serialization

# Load private key
with open('private_key.p8', 'rb') as key:
 private_key = serialization.load_pem_private_key(
 key.read(),
 password=None
 )

# Connection with key pair
conn_params = {
 'user': 'your_username',
 'account': 'your_account',
 'private_key': private_key,
 'warehouse': 'COMPUTE_WH',
 'database': 'YOUR_DB'
}
```

SAP HANA Integration

01

Installation Setup

```
# Install HANA client
pip install hdbcli
pip install sqlalchemy-hana

# Alternative: PyHDB
pip install pyhdb
```

Multiple connection options available depending on your HANA version and requirements.

02

Connection Methods

```
from sqlalchemy import
create_engine

# SQLAlchemy connection
engine = create_engine(

'hana://username:password@hostna
me:30015'
)

# Direct hdbcli connection
from hdbcli import dbapi
conn = dbapi.connect(
  address='hostname',
  port=30015,
  user='username',
  password='password'
)
```

SAP HANA Integration

01

Query Execution

```
# Read from HANA table
df = pd.read_sql("""
SELECT * FROM schema.table_name
WHERE date_column > ADD_DAYS(CURRENT_DATE, -30)
""", engine)

# Write to HANA
df.to_sql('target_table', engine,
schema='ANALYTICS',
if_exists='append',
method='multi')
```

Database Connection Management

Security Best Practices

- Store credentials in environment variables
- Use connection pooling for performance
- Implement proper error handling
- Enable SSL/TLS encryption
- Regularly rotate access credentials

Connection Configuration

```
import os
from sqlalchemy import
create_engine
from sqlalchemy.pool import
QueuePool

# Environment-based config
db_url =
f"snowflake://{os.getenv('USER')}:" \
        f"{os.getenv('PASSWORD')}@" \
        f"{os.getenv('ACCOUNT')}"

engine = create_engine(db_url,
                       poolclass=QueuePool,
                       pool_size=5,
                       max_overflow=10)
```

Error Handling

```
import pandas as pd
from sqlalchemy.exc import
SQLAlchemyError

try:
 df = pd.read_sql(query, engine)
except SQLAlchemyError as e:
 print(f"Database error: {e}")
 # Implement retry logic
except Exception as e:
 print(f"General error: {e}")
 # Log error details
finally:
 engine.dispose()
```

Advanced File Operations

1 Chunked Processing

Handle large files that exceed memory limits by processing in manageable chunks.

```
chunk_list = []
for chunk in pd.read_csv('large_file.csv',
chunksize=10000):
# Process each chunk
processed_chunk = chunk.groupby('category').sum()
chunk_list.append(processed_chunk)

# Combine results
final_df = pd.concat(chunk_list, ignore_index=True)
```

2 Compression Handling

Automatically detect and handle compressed files for efficient storage and transfer.

```
# Automatic compression detection
df = pd.read_csv('data.csv.gz')  # gzip
df = pd.read_csv('data.csv.bz2') # bzip2
df = pd.read_csv('data.csv.xz')  # xz

# Explicit compression
df.to_csv('output.csv.gz', compression='gzip')
df.to_parquet('output.parquet.snappy',
compression='snappy')
```

3 Multi-file Processing

Process multiple files efficiently using glob patterns and concatenation strategies.

```
import glob

# Read multiple CSV files
file_list = glob.glob('data_*.csv')
df_list = []

for file in file_list:
temp_df = pd.read_csv(file)
temp_df['source_file'] = file
df_list.append(temp_df)

combined_df = pd.concat(df_list, ignore_index=True)
```

Data Type Optimization

Memory Efficiency Strategies

Optimize DataFrame memory usage through intelligent data type selection and conversion techniques.

```
# Check memory usage
print(df.info(memory_usage='deep'))

# Optimize numeric types
df['int_col'] = pd.to_numeric(df['int_col'], downcast='integer')
df['float_col'] = pd.to_numeric(df['float_col'], downcast='float')

# Convert to category for repeated strings
df['category_col'] = df['category_col'].astype('category')

# Use sparse arrays for mostly null data
df['sparse_col'] = df['sparse_col'].astype('Sparse[float64]')
```

These optimizations can reduce memory usage by 50-80% in typical datasets.

Memory Usage Comparison

Data Type	Memory (MB)
object (default)	45.2
category	12.1
int64	8.0
int32	4.0
float64	8.0
float32	4.0

Always profile your data to determine optimal type conversions for your specific use case.

Real-World Example: Sales Data Pipeline

1

Data Ingestion

```
# Read from multiple sources
sales_csv = pd.read_csv('sales_2024.csv',
                        parse_dates=['transaction_date'])
customer_excel = pd.read_excel('customers.xlsx')

# Query from Snowflake
inventory_query = """
SELECT product_id, stock_level, last_updated
FROM inventory.current_stock
WHERE last_updated >= CURRENT_DATE - 7
"""
inventory_df = pd.read_sql(inventory_query, snowflake_engine)
```

2

Data Processing

```
# Merge datasets
merged_df = sales_csv.merge(customer_excel, on='customer_id', how='left')
merged_df = merged_df.merge(inventory_df, on='product_id', how='left')

# Date-based analysis
merged_df['month'] = merged_df['transaction_date'].dt.month
merged_df['quarter'] = merged_df['transaction_date'].dt.quarter

# Calculate metrics
daily_sales = merged_df.groupby('transaction_date').agg({
    'amount': 'sum',
    'transaction_id': 'count'
}).rename(columns={'transaction_id': 'transaction_count'})
```

3

Output Generation

```
# Export to multiple formats
daily_sales.to_csv('daily_sales_report.csv')
daily_sales.to_excel('daily_sales_report.xlsx', sheet_name='Daily Summary')
daily_sales.to_parquet('daily_sales_report.parquet')

# Write back to database
daily_sales.to_sql('daily_sales_summary', snowflake_engine,
 if_exists='replace', index=True)
```

Error Handling and Validation

1

File Existence Checks

```
import os
from pathlib import Path

def safe_read_csv(filepath):
    if Path(filepath).exists():
        try:
            return
pd.read_csv(filepath)
        except
pd.errors.EmptyDataError:
            print(f"Warning:
{filepath} is empty")
            return pd.DataFrame()
    else:
        raise
FileNotFoundError(f"File
{filepath} not found")
```

2

Data Validation

```
def validate_dataframe(df,
required_columns):
    # Check for required columns
    missing_cols =
set(required_columns) -
set(df.columns)
    if missing_cols:
        raise ValueError(f"Missing
columns: {missing_cols}")

    # Check for empty DataFrame
    if df.empty:
        raise
ValueError("DataFrame is
empty")

    return True
```

3

Robust Date Parsing

```
def parse_dates_safely(df, date_columns):
 for col in date_columns:
 try:
 df[col] = pd.to_datetime(df[col], errors='coerce')
 null_count = df[col].isnull().sum()
 if null_count > 0:
 print(f"Warning: {null_count} invalid dates in {col}")
 except Exception as e:
 print(f"Error parsing {col}: {e}")
 return df
```

Performance Monitoring and Profiling

Timing Operations

```
import time
from functools import wraps

def time_operation(func):
    @wraps(func)
    def wrapper(*args, **kwargs):
        start_time = time.time()
        result = func(*args, **kwargs)
        end_time = time.time()
        print(f"{func.__name__} took {end_time - start_time:.2f}
seconds")
        return result
    return wrapper

@time_operation
def process_large_file(filepath):
    return pd.read_csv(filepath, chunksize=10000)
```

Memory Profiling

```
# Monitor memory usage
def check_memory_usage(df, operation_name):
    memory_mb = df.memory_usage(deep=True).sum() / (1024 *
1024)
    print(f"{operation_name}: {memory_mb:.2f} MB")

# Before and after comparisons
check_memory_usage(df, "Before optimization")
df_optimized = df.astype({'category_col': 'category'})
check_memory_usage(df_optimized, "After optimization")
```

Regular profiling helps identify bottlenecks and optimization opportunities in your data pipeline.

Configuration Management

Environment Configuration

```
import os
from configparser import
ConfigParser

# Using environment variables
DATABASE_URL =
os.getenv('DATABASE_URL')
API_KEY = os.getenv('API_KEY')

# Using config files
config = ConfigParser()
config.read('config.ini')

snowflake_config = {
    'user': config.get('snowflake', 'user'),
    'password': config.get('snowflake',
'password'),
    'account': config.get('snowflake',
'account'),
    'warehouse': config.get('snowflake',
'warehouse')
}
```

Flexible File Paths

```
from pathlib import Path

# Define base paths
DATA_DIR = Path('./data')
OUTPUT_DIR = Path('./output')
CONFIG_DIR = Path('./config')

# Ensure directories exist
DATA_DIR.mkdir(exist_ok=True)
OUTPUT_DIR.mkdir(exist_ok=True)

# Build file paths dynamically
input_file = DATA_DIR /
f"sales_{date}.csv"
output_file = OUTPUT_DIR /
f"processed_{date}.parquet"
```

Logging Setup

```
import logging

# Configure logging
logging.basicConfig(
 level=logging.INFO,
 format='%(asctime)s - %(levelname)s -
%(message)s',
 handlers=[

logging.FileHandler('data_pipeline.log'
),
 logging.StreamHandler()
 ]
)

logger = logging.getLogger(__name__)
logger.info("Starting data processing
pipeline")
```

Testing Your Data Pipeline

1 Unit Testing Data Functions

```
import unittest
import pandas as pd

class
TestDataProcessing(unittest.TestCase):

    def setUp(self):
        self.sample_data =
pd.DataFrame({
            'date': ['2024-01-01', '2024-
01-02'],
            'amount': [100, 200],
            'category': ['A', 'B']
        })

    def test_date_parsing(self):
        df =
parse_dates_safely(self.sample_data, ['date'])

self.assertTrue(pd.api.types.is_datetime64_any_dtype(df['date']))

    def test_data_validation(self):
        required_cols = ['date',
'amount', 'category']

self.assertTrue(validate_dataframe(self.sample_data, required_cols))
```

2 Integration Testing

```
def test_database_connection():
    """Test database connectivity and
basic query execution"""
    try:
        test_query = "SELECT 1 as
test_column"
        result = pd.read_sql(test_query,
engine)
        assert len(result) == 1
        assert result.iloc[0, 0] == 1
        print("Database connection
test passed")
    except Exception as e:
        print(f"Database connection
test failed: {e}")
        raise
```

3 Data Quality Checks

```
def run_data_quality_checks(df):
 """Comprehensive data quality
validation"""
 checks = {
 'no_duplicate_rows': len(df) ==
len(df.drop_duplicates()),
 'no_all_null_columns': not
df.isnull().all().any(),
 'expected_row_count': len(df) > 0,
 'valid_date_range': df['date'].min()
> pd.Timestamp('2020-01-01')
 }

 failed_checks = [check for check,
passed in checks.items() if not
passed]
 if failed_checks:
 raise ValueError(f"Quality checks
failed: {failed_checks}")

 return True
```

Key Takeaways

Date Mastery

Leverage pandas datetime functionality for robust temporal analysis. Use timezone-aware operations and efficient date parsing strategies for enterprise-grade data processing.

Format Flexibility

Choose the right file format for your use case. CSV for compatibility, Parquet for performance, Excel for business users, and JSON for semi-structured data.

Database Integration

Master Snowflake and SAP HANA connections for seamless enterprise data access. Implement proper security, error handling, and performance optimization techniques.

Performance Focus

Profile your operations, optimize data types, and implement chunked processing for large datasets. Monitor memory usage and processing times to maintain efficient pipelines.

These techniques form the foundation of professional data engineering workflows. Practice with real datasets and gradually build more complex integration patterns.