Esercitazione: Parsing CSV in Python

Obiettivi dell'esercitazione

Al termine di questa esercitazione, lo studente sarà in grado di:

- Utilizzare il modulo `csv` della libreria standard Python per leggere e scrivere file CSV
- Gestire diverse varianti del formato CSV (delimitatori, encoding, formati)
- Implementare validazione e conversione dei tipi di dato
- Processare file CSV di grandi dimensioni in modo efficiente
- Gestire errori e casi limite nel parsing CSV
- Applicare operazioni di analisi e trasformazione su dati tabulari

Prerequisiti

- Conoscenza di base di Python (variabili, strutture di controllo, funzioni)
- Familiarità con file I/O in Python
- Comprensione dei concetti di liste e dizionari
- Conoscenza base del formato CSV

Materiale necessario

- Python 3.8 o superiore
- Editor di testo o IDE (VS Code, PyCharm, o simili)
- File CSV forniti per gli esercizi (vedi sezione "Preparazione dei dati")

Parte 1: Preparazione dei dati

Prima di iniziare gli esercizi, create i seguenti file CSV nella vostra directory di lavoro:

File 1: `studenti.csv`

```
matricola,nome,cognome,corso,anno,media_voti,crediti
S001,Mario,Rossi,Informatica,3,27.5,150
S002,Laura,Bianchi,Informatica,2,28.9,120
S003,Giovanni,Verdi,Elettronica,1,24.3,60
S004,Francesca,Neri,Informatica,3,29.1,165
S005,Alessandro,Blu,Telecomunicazioni,2,26.7,110
S006,Sofia,Gialli,Informatica,1,30.0,75
S007,Luca,Viola,Elettronica,3,25.8,140
S008,Chiara,Rosa,Informatica,2,27.2,125
```

File 2: `vendite.csv` (formato europeo con punto e virgola)

```
data;prodotto;categoria;quantità;prezzo_unitario;cliente
2024-01-15;Laptop Dell XPS;Elettronica;2;1299.99;Rossi Mario
2024-01-16;Mouse Logitech;Accessori;5;29.50;Bianchi Laura
2024-01-16;Tastiera Meccanica;Accessori;3;89.99;Verdi Giovanni
2024-01-17;Monitor Samsung 27";Elettronica;1;349.00;Neri Francesca
2024-01-18;Webcam HD;Accessori;4;59.99;Blu Alessandro
2024-01-19;Laptop HP Pavilion;Elettronica;1;899.99;Gialli Sofia
2024-01-20;Cuffie Wireless;Accessori;6;79.50;Viola Luca
```

File 3: `sensori.csv` (dati con valori mancanti e problematici)

```
timestamp,sensore_id,temperatura,umidità,pressione,stato
2024-11-19 08:00:00,SENS01,22.5,65.3,1013.2,OK
2024-11-19 08:05:00,SENS01,22.7,65.1,1013.1,OK
2024-11-19 08:10:00,SENS01,NULL,64.9,1013.0,ERROR
2024-11-19 08:15:00,SENS01,23.1,NULL,1012.9,WARNING
2024-11-19 08:00:00,SENS02,21.8,68.2,1013.3,OK
2024-11-19 08:05:00,SENS02,22.0,68.0,INVALID,ERROR
2024-11-19 08:10:00,SENS02,22.2,67.8,1013.1,OK
```

Parte 2: Esercizi guidati

Esercizio 1: Lettura base con csv.reader (★☆☆☆☆)

Obiettivo: Familiarizzare con l'API base del modulo csv.

Consegna: Scrivere un programma che legga il file `studenti.csv` e stampi a video:

1. Il numero totale di studenti
2. Tutti i dati di ogni studente in formato leggibile

Template di partenza:

```python
import csv

def leggi_studenti_base(filename):
    """
    Legge il file CSV e stampa le informazioni degli studenti.

    Args:
        filename: percorso del file CSV
    """
    # TODO: Implementare la funzione
    pass

if __name__ == "__main__":
    leggi_studenti_base('studenti.csv')
```

Output atteso:

```
Totale studenti: 8

Studente 1:
  Matricola: S001
  Nome: Mario Rossi
  Corso: Informatica
  Anno: 3
  Media voti: 27.5
  Crediti: 150

[...continua per tutti gli studenti...]
```

Suggerimenti:

- Usare `with open()` per garantire la chiusura del file
- Ricordarsi di specificare `encoding='utf-8'` e `newline=''`
- La prima riga contiene gli header

Criteri di valutazione:

- Corretta apertura e chiusura del file: 2 punti
- Lettura corretta di tutti i record: 3 punti
- Formattazione output leggibile: 2 punti
- Gestione degli header: 3 punti

Esercizio 2: Lettura strutturata con DictReader (★★☆☆☆)

Obiettivo: Utilizzare DictReader per un accesso più semantico ai dati.

Consegna: Scrivere una funzione che:

1. Legga il file `studenti.csv` usando `DictReader`
2. Restituisca una lista di dizionari contenente solo gli studenti di Informatica
3. Calcoli e stampi la media dei voti degli studenti di Informatica

Template di partenza:

```
import csv

def analizza_studenti_informatica(filename):
    """
    Estrae gli studenti di Informatica e calcola statistiche.

    Args:
        filename: percorso del file CSV
```

```
    Returns:
        tuple: (lista_studenti, media_voti)
    """
    studenti_informatica = []

    # TODO: Implementare la lettura con DictReader
    # TODO: Filtrare solo studenti di Informatica
    # TODO: Calcolare la media dei voti

    return studenti_informatica, media_voti

def stampa_risultati(studenti, media):
    """Stampa i risultati in formato leggibile."""
    # TODO: Implementare
    pass

if __name__ == "__main__":
    studenti, media = analizza_studenti_informatica('studenti.csv')
    stampa_risultati(studenti, media)
```

Output atteso:

```
Studenti di Informatica: 5

Matricola  Nome              Media Voti  Crediti
S001       Mario Rossi       27.5        150
S002       Laura Bianchi     28.9        120
S004       Francesca Neri    29.1        165
S006       Sofia Gialli      30.0        75
S008       Chiara Rosa       27.2        125

Media voti del corso: 28.54
```

Suggerimenti:

- Usare l'accesso per chiave: `row['campo']`
- Convertire i valori numerici da stringa: `float(row['media_voti'])`
- Usare una list comprehension per filtrare

Criteri di valutazione:

- Uso corretto di DictReader: 3 punti
- Filtro corretto per corso: 3 punti
- Calcolo corretto della media: 2 punti
- Formattazione output tabellare: 2 punti

Esercizio 3: Gestione di formati CSV alternativi (★★☆☆☆)

Obiettivo: Lavorare con delimitatori e formati non standard.

Consegna: Scrivere un programma che:

1. Legga il file `vendite.csv` (delimitatore punto e virgola)
2. Calcoli il fatturato totale per ogni categoria di prodotto
3. Identifichi il cliente che ha speso di più

Template di partenza:

```
import csv
from collections import defaultdict

def analizza_vendite(filename):
    """
    Analizza i dati di vendita.

    Args:
        filename: percorso del file CSV

    Returns:
        tuple: (fatturato_per_categoria, miglior_cliente, spesa_cliente)
    """
    fatturato = defaultdict(float)
    spesa_clienti = defaultdict(float)

    # TODO: Leggere il CSV con delimitatore ';'
    # TODO: Calcolare fatturato per categoria
    # TODO: Calcolare spesa per cliente
    # TODO: Trovare il cliente che ha speso di più

    return dict(fatturato), miglior_cliente, max_spesa

if __name__ == "__main__":
    fatturato, cliente, spesa = analizza_vendite('vendite.csv')

    print("Fatturato per categoria:")
    for categoria, totale in sorted(fatturato.items()):
        print(f"  {categoria}: €{totale:.2f}")

    print(f"\nMiglior cliente: {cliente} (€{spesa:.2f})")
```

Output atteso:

```
Fatturato per categoria:
  Accessori: €924.44
  Elettronica: €3898.98

Miglior cliente: Rossi Mario (€2599.98)
```

Suggerimenti:

- Specificare `delimiter=';'` in DictReader
- Calcolare l'importo: `quantità * prezzo_unitario`
- Usare `defaultdict` per accumulare valori

Criteri di valutazione:

- Configurazione corretta del delimiter: 2 punti
- Calcolo corretto del fatturato per categoria: 4 punti
- Identificazione corretta del miglior cliente: 3 punti
- Formattazione corretta dei valori monetari: 1 punto

Esercizio 4: Conversione dei tipi e validazione (★★★☆☆)

Obiettivo: Implementare conversione automatica dei tipi con gestione degli errori.

Consegna: Creare un sistema di parsing CSV con conversione automatica dei tipi che:

1. Legga `sensori.csv`
2. Converta automaticamente i campi numerici in float
3. Gestisca valori mancanti (NULL, stringhe vuote) e valori invalidi
4. Produca un report delle letture valide e degli errori

Template di partenza:

```
import csv
from datetime import datetime
from typing import Dict, List, Any, Optional

class CSVParserConValidazione:
    """Parser CSV con conversione tipi e validazione."""

    def __init__(self, filename: str, type_converters: Dict[str,
callable]):
        """
        Inizializza il parser.

        Args:
            filename: percorso del file CSV
            type_converters: dizionario {campo: funzione_conversione}
        """
        self.filename = filename
        self.type_converters = type_converters
        self.errori = []

    def parse(self) -> List[Dict[str, Any]]:
        """
        Parsa il file CSV applicando le conversioni.

        Returns:
            Lista di record con valori convertiti
        """
```

```
        records = []

        # TODO: Implementare il parsing con gestione errori
        # TODO: Per ogni record, applicare i convertitori
        # TODO: Gestire eccezioni di conversione
        # TODO: Registrare gli errori in self.errori

        return records

    def get_report_errori(self) -> str:
        """Genera un report degli errori incontrati."""
        # TODO: Implementare
        pass

def converti_timestamp(value: str) -> datetime:
    """Converte una stringa in datetime."""
    # TODO: Implementare
    pass

def converti_float_safe(value: str) -> Optional[float]:
    """
    Converte una stringa in float, restituendo None per valori invalidi.
    """
    # TODO: Implementare
    # Gestire 'NULL', stringhe vuote, 'INVALID', etc.
    pass

if __name__ == "__main__":
    # Definisci i convertitori per ogni campo
    converters = {
        'timestamp': converti_timestamp,
        'temperatura': converti_float_safe,
        'umidità': converti_float_safe,
        'pressione': converti_float_safe
    }

    parser = CSVParserConValidazione('sensori.csv', converters)
    dati = parser.parse()

    # Stampa statistiche
    print(f"Record letti: {len(dati)}")
    print(f"Record con errori: {len(parser.errori)}")
    print("\n" + parser.get_report_errori())

    # Calcola temperatura media delle letture valide
    temperature_valide = [r['temperatura'] for r in dati
                          if r.get('temperatura') is not None]
    if temperature_valide:
        media_temp = sum(temperature_valide) / len(temperature_valide)
        print(f"\nTemperatura media: {media_temp:.2f}°C")
```

Output atteso:

```
Record letti: 7
Record con errori: 3

Errori di conversione:
  Riga 3: Campo 'temperatura' - valore 'NULL' non valido
  Riga 4: Campo 'umidità' - valore 'NULL' non valido
  Riga 6: Campo 'pressione' - valore 'INVALID' non valido

Temperatura media: 22.38°C
```

Suggerimenti:

- Usare `try/except` per catturare errori di conversione
- `datetime.strptime()` per parsing delle date
- Considerare `None` come valore per dati mancanti
- Tenere traccia del numero di riga per il report errori

Criteri di valutazione:

- Implementazione corretta dei convertitori: 4 punti
- Gestione robusta degli errori: 4 punti
- Report errori informativo: 2 punti
- Calcolo corretto delle statistiche: 2 punti

Esercizio 5: Scrittura CSV con trasformazione dati (★★★☆☆)

Obiettivo: Leggere, trasformare e scrivere dati CSV.

Consegna: Creare un programma che:

1. Legga `studenti.csv`
2. Calcoli per ogni studente:
   - Media ponderata CFU (assumendo stesso peso per ogni esame)
   - Percentuale di completamento del corso (su 180 CFU totali)
   - Classificazione: "Eccellente" (≥28), "Buono" (≥25), "Sufficiente" (<25)
3. Generi un nuovo file `studenti_elaborati.csv` con questi dati aggiuntivi
4. Generi un file `statistiche_corsi.csv` con statistiche aggregate per corso

Template di partenza:

```python
import csv
from collections import defaultdict

class ElaboratoreStudenti:
    """Elabora dati studenti e genera report."""

    CFU_TOTALI = 180
```

```
    def __init__(self, input_file: str):
        self.input_file = input_file
        self.studenti = []

    def carica_dati(self):
        """Carica i dati dal file CSV."""
        # TODO: Implementare
        pass

    def elabora_studente(self, studente: dict) -> dict:
        """
        Elabora i dati di un singolo studente.

        Args:
            studente: dizionario con dati studente

        Returns:
            dizionario con dati originali + campi calcolati
        """
        elaborato = studente.copy()

        # TODO: Calcolare percentuale_completamento
        # TODO: Determinare classificazione
        # TODO: Aggiungere campi al dizionario

        return elaborato

    def salva_studenti_elaborati(self, output_file: str):
        """Salva i dati elaborati in un nuovo CSV."""
        # TODO: Implementare usando csv.DictWriter
        pass

    def calcola_statistiche_corsi(self) -> dict:
        """
        Calcola statistiche aggregate per corso.

        Returns:
            dizionario {corso: {'studenti': n, 'media_voti': x, ...}}
        """
        stats = defaultdict(lambda: {
            'studenti': 0,
            'media_voti': 0,
            'media_crediti': 0
        })

        # TODO: Calcolare statistiche per ogni corso

        return dict(stats)

    def salva_statistiche(self, output_file: str):
        """Salva le statistiche in un CSV."""
        # TODO: Implementare
        pass
```

```
if __name__ == "__main__":
    elaboratore = ElaboratoreStudenti('studenti.csv')
    elaboratore.carica_dati()

    # Genera file con studenti elaborati
    elaboratore.salva_studenti_elaborati('studenti_elaborati.csv')

    # Genera file con statistiche
    elaboratore.salva_statistiche('statistiche_corsi.csv')

    print("Elaborazione completata!")
    print("File generati:")
    print("  - studenti_elaborati.csv")
    print("  - statistiche_corsi.csv")
```

File output attesi:

`studenti_elaborati.csv`:

```
matricola,nome,cognome,corso,anno,media_voti,crediti,percentuale_completam
ento,classificazione
S001,Mario,Rossi,Informatica,3,27.5,150,83.33,Eccellente
S002,Laura,Bianchi,Informatica,2,28.9,120,66.67,Eccellente
[...etc...]
```

`statistiche_corsi.csv`:

```
corso,numero_studenti,media_voti,media_crediti
Elettronica,2,25.05,100.00
Informatica,5,28.54,127.00
Telecomunicazioni,1,26.70,110.00
```

Criteri di valutazione:

- Lettura corretta dei dati: 2 punti
- Calcoli corretti (percentuale, classificazione): 4 punti
- Scrittura corretta file studenti elaborati: 3 punti
- Calcolo e scrittura statistiche aggregate: 3 punti
- Struttura object-oriented del codice: 3 punti

Parte 3: Esercizi avanzati

Esercizio 6: Processing streaming di file grandi (★★★★☆)

Obiettivo: Implementare elaborazione efficiente di file CSV di grandi dimensioni senza caricarli completamente in memoria.

Contesto: Vi viene fornito un file `transazioni.csv` contenente 1 milione di transazioni (dovete generarlo per il test). Il file è troppo grande per essere caricato interamente in memoria.

Consegna: Implementare un sistema che:

1. Generi un file di test con 1 milione di transazioni
2. Calcoli statistiche aggregate (totale per categoria, transazione massima) usando approccio streaming
3. Confronti tempo di esecuzione e memoria usata tra approccio "carica tutto" e streaming

Template di partenza:

```
import csv
import random
from datetime import datetime, timedelta
import time
import tracemalloc

class GeneratoreTransazioni:
    """Genera file CSV di test con transazioni."""

    CATEGORIE = ['Elettronica', 'Abbigliamento', 'Alimentari',
                 'Casa', 'Sport', 'Libri']

    @staticmethod
    def genera_file(filename: str, num_records: int):
        """
        Genera un file CSV con transazioni casuali.

        Args:
            filename: nome del file da creare
            num_records: numero di transazioni da generare
        """
        print(f"Generazione di {num_records:,} transazioni...")

        # TODO: Implementare generazione file
        # Campi: id, data, categoria, importo, cliente_id
        # Usare csv.writer per scrivere efficacemente

        pass

class AnalizzatoreTransazioni:
    """Analizza transazioni con approccio streaming."""

    def __init__(self, filename: str):
        self.filename = filename

    def analizza_streaming(self):
        """
        Analizza il file usando approccio streaming.
        Calcola statistiche senza caricare tutto in memoria.

        Returns:
```

```
            dict con statistiche aggregate
        """
        stats = {
            'totale_per_categoria': {},
            'transazione_max': {'importo': 0, 'id': None},
            'num_transazioni': 0,
            'importo_totale': 0
        }

        # TODO: Implementare analisi streaming
        # Leggere una riga alla volta
        # Aggiornare statistiche incrementalmente

        return stats

    def analizza_caricamento_completo(self):
        """
        Analizza il file caricandolo completamente in memoria.
        (Approccio inefficiente per confronto)
        """
        # TODO: Implementare (carica tutto, poi analizza)
        pass

def confronta_approcci(filename: str):
    """Confronta prestazioni dei due approcci."""

    analizzatore = AnalizzatoreTransazioni(filename)

    # Test approccio streaming
    print("\n=== Approccio Streaming ===")
    tracemalloc.start()
    start_time = time.time()

    stats_streaming = analizzatore.analizza_streaming()

    tempo_streaming = time.time() - start_time
    memoria_streaming = tracemalloc.get_traced_memory()[1] / 1024 / 1024
# MB
    tracemalloc.stop()

    # TODO: Ripetere per approccio caricamento completo

    # TODO: Stampare confronto prestazioni

    return stats_streaming

if __name__ == "__main__":
    filename = 'transazioni_test.csv'

    # Genera file di test (ridurre a 100k per test veloci)
    GeneratoreTransazioni.genera_file(filename, 100000)

    # Confronta approcci
    stats = confronta_approcci(filename)
```

```
    print("\n=== Statistiche ===")
    print(f"Totale transazioni: {stats['num_transazioni']:,}")
    print(f"Importo totale: €{stats['importo_totale']:,.2f}")
    print("\nTotale per categoria:")
    for cat, tot in sorted(stats['totale_per_categoria'].items()):
        print(f"  {cat}: €{tot:,.2f}")
```

Output atteso:

```
Generazione di 100,000 transazioni...
File generato: transazioni_test.csv (4.2 MB)

=== Approccio Streaming ===
Tempo: 0.82s
Memoria di picco: 2.1 MB

=== Approccio Caricamento Completo ===
Tempo: 1.34s
Memoria di picco: 45.7 MB

=== Statistiche ===
Totale transazioni: 100,000
Importo totale: €4,987,234.56
[...etc...]
```

Criteri di valutazione:

- Generazione corretta del file di test: 3 punti
- Implementazione corretta streaming: 5 punti
- Calcolo corretto delle statistiche: 4 punti
- Confronto prestazioni con misurazioni: 3 punti

Esercizio 7: Parser CSV robusto custom (★★★★★)

Obiettivo: Implementare un parser CSV da zero che gestisca correttamente l'RFC 4180.

Consegna: Implementare una classe `CustomCSVParser` che:

1. Implementi un automa a stati finiti per il parsing
2. Gestisca correttamente campi quoted con virgole e newline
3. Gestisca escape di doppi apici
4. Fornisca diagnostica dettagliata degli errori

Template di partenza:

```
from enum import Enum, auto
from typing import List, Iterator, Optional
```

```python
from dataclasses import dataclass

class ParseState(Enum):
    """Stati dell'automa a stati finiti."""
    FIELD_START = auto()
    IN_FIELD = auto()
    IN_QUOTED_FIELD = auto()
    QUOTE_IN_QUOTED_FIELD = auto()

@dataclass
class ParseError:
    """Rappresenta un errore di parsing."""
    line: int
    column: int
    message: str
    context: str

class CustomCSVParser:
    """
    Parser CSV RFC 4180 compliant implementato come automa a stati finiti.
    """

    def __init__(self, delimiter: str = ',', quotechar: str = '"',
                 doublequote: bool = True):
        self.delimiter = delimiter
        self.quotechar = quotechar
        self.doublequote = doublequote
        self.errors: List[ParseError] = []

    def parse_line(self, line: str, line_number: int = 1) ->
Optional[List[str]]:
        """
        Parsa una singola riga CSV usando automa a stati finiti.

        Args:
            line: riga da parsare
            line_number: numero di riga (per error reporting)

        Returns:
            Lista di campi o None se errore fatale
        """
        fields = []
        current_field = []
        state = ParseState.FIELD_START

        # TODO: Implementare automa a stati finiti
        # Gestire tutte le transizioni di stato
        # Registrare errori in self.errors

        return fields

    def parse_file(self, filename: str, encoding: str = 'utf-8') ->
Iterator[List[str]]:
        """
```

```
        Parsa un file CSV gestendo record multi-riga.

        Args:
            filename: percorso del file
            encoding: encoding del file

        Yields:
            Lista di campi per ogni record
        """
        # TODO: Implementare
        # Gestire record che si estendono su più righe
        # Mantenere conteggio righe per error reporting
        pass

    def get_error_report(self) -> str:
        """Genera report dettagliato degli errori."""
        # TODO: Implementare
        pass

# Test cases
def test_parser():
    """Test suite per il parser custom."""

    parser = CustomCSVParser()

    # Test 1: CSV semplice
    result = parser.parse_line('campo1,campo2,campo3')
    assert result == ['campo1', 'campo2', 'campo3'], "Test 1 fallito"

    # Test 2: Campi quoted con virgole
    result = parser.parse_line('"campo, con virgola",campo2')
    assert result == ['campo, con virgola', 'campo2'], "Test 2 fallito"

    # Test 3: Doppi apici escapati
    result = parser.parse_line('"campo con ""quote""",campo2')
    assert result == ['campo con "quote"', 'campo2'], "Test 3 fallito"

    # Test 4: Campi vuoti
    result = parser.parse_line('campo1,,campo3,')
    assert len(result) == 4 and result[1] == '', "Test 4 fallito"

    # TODO: Aggiungere più test

    print("Tutti i test passati!")

if __name__ == "__main__":
    test_parser()

    # Test con file reale
    parser = CustomCSVParser(delimiter=';')

    for record in parser.parse_file('vendite.csv'):
        print(record)
```

```
    if parser.errors:
        print("\n" + parser.get_error_report())
```

Criteri di valutazione:

- Implementazione corretta automa a stati: 8 punti
- Gestione corretta di tutti i casi limite: 6 punti
- Error reporting dettagliato: 3 punti
- Test suite completa: 3 punti

Parte 4: Progetto finale integrativo

Progetto: Sistema di analisi dati corso universitario (★★★★★)

Obiettivo: Integrare tutte le competenze acquisite in un progetto completo.

Scenario: Siete incaricati di sviluppare un sistema di analisi per il coordinamento del corso di Informatica. Il sistema deve processare diversi file CSV e generare report completi.

File di input forniti:

1. `studenti.csv` - anagrafica studenti
2. `esami.csv` - storico esami sostenuti
3. `insegnamenti.csv` - informazioni su corsi e docenti

Requisiti funzionali:

1. **Caricamento dati multi-file**:
    - Leggere e validare tutti i file CSV
    - Gestire inconsistenze e dati mancanti
    - Generare log degli errori
2. **Analisi per studente**:
    - Calcolare media pesata per CFU
    - Determinare CFU conseguiti per anno
    - Identificare esami mancanti rispetto al piano di studi
    - Stimare data di laurea
3. **Analisi per insegnamento**:
    - Media voti per ogni insegnamento
    - Tasso di successo (promossi/totali)
    - Distribuzione voti
    - Confronto anno su anno
4. **Report aggregati**:
    - Classifica studenti per media

- Insegnamenti più difficili
- Statistiche di coorte
- Trend temporali

5. **Export risultati**:
   - Generare CSV con dati elaborati
   - Generare report in formato Markdown
   - Creare file JSON per integrazione con altri sistemi

Struttura del progetto:

```
progetto_analisi/
├── data/
│   ├── input/
│   │   ├── studenti.csv
│   │   ├── esami.csv
│   │   └── insegnamenti.csv
│   └── output/
│       ├── studenti_elaborati.csv
│       ├── statistiche_insegnamenti.csv
│       ├── report_generale.md
│       └── dati_export.json
├── src/
│   ├── __init__.py
│   ├── data_loader.py
│   ├── validators.py
│   ├── analyzers.py
│   ├── report_generator.py
│   └── main.py
├── tests/
│   ├── test_loader.py
│   ├── test_validators.py
│   └── test_analyzers.py
├── requirements.txt
└── README.md
```

Template iniziale (`src/main.py`):

```python
"""
Sistema di Analisi Dati Corso Universitario
"""

import csv
import json
from pathlib import Path
from typing import Dict, List, Any
from dataclasses import dataclass, asdict
from datetime import datetime
```

```python
@dataclass
class Studente:
    """Rappresenta uno studente."""
    matricola: str
    nome: str
    cognome: str
    anno_immatricolazione: int
    corso: str

@dataclass
class Esame:
    """Rappresenta un esame sostenuto."""
    matricola: str
    codice_insegnamento: str
    data_esame: datetime
    voto: int
    cfu: int

@dataclass
class Insegnamento:
    """Rappresenta un insegnamento."""
    codice: str
    nome: str
    docente: str
    cfu: int
    anno: int
    semestre: int

class DataLoader:
    """Gestisce il caricamento dei dati CSV."""

    def __init__(self, data_dir: Path):
        self.data_dir = data_dir
        self.errors = []

    def carica_studenti(self) -> List[Studente]:
        """Carica e valida gli studenti."""
        # TODO: Implementare
        pass

    def carica_esami(self) -> List[Esame]:
        """Carica e valida gli esami."""
        # TODO: Implementare
        pass

    def carica_insegnamenti(self) -> List[Insegnamento]:
        """Carica e valida gli insegnamenti."""
        # TODO: Implementare
        pass

class Analyzer:
    """Esegue analisi sui dati."""

    def __init__(self, studenti: List[Studente],
```

```
                 esami: List[Esame],
                 insegnamenti: List[Insegnamento]):
        self.studenti = studenti
        self.esami = esami
        self.insegnamenti = insegnamenti

    def analizza_studente(self, matricola: str) -> Dict[str, Any]:
        """Analizza i dati di uno specifico studente."""
        # TODO: Implementare
        pass

    def analizza_insegnamento(self, codice: str) -> Dict[str, Any]:
        """Analizza i dati di uno specifico insegnamento."""
        # TODO: Implementare
        pass

    def genera_classifica(self) -> List[Dict[str, Any]]:
        """Genera classifica studenti per media."""
        # TODO: Implementare
        pass

    def insegnamenti_piu_difficili(self, n: int = 5) -> List[Dict[str,
Any]]:
        """Identifica gli N insegnamenti più difficili."""
        # TODO: Implementare
        pass

class ReportGenerator:
    """Genera report in vari formati."""

    def __init__(self, output_dir: Path):
        self.output_dir = output_dir

    def genera_csv(self, data: List[Dict], filename: str):
        """Genera file CSV."""
        # TODO: Implementare
        pass

    def genera_markdown(self, analyzer: Analyzer, filename: str):
        """Genera report in Markdown."""
        # TODO: Implementare
        pass

    def genera_json(self, data: Dict, filename: str):
        """Genera export JSON."""
        # TODO: Implementare
        pass

def main():
    """Funzione principale."""

    # Setup paths
    base_dir = Path(__file__).parent.parent
    data_dir = base_dir / 'data' / 'input'
```

```
    output_dir = base_dir / 'data' / 'output'
    output_dir.mkdir(exist_ok=True)

    # Carica dati
    print("Caricamento dati...")
    loader = DataLoader(data_dir)
    studenti = loader.carica_studenti()
    esami = loader.carica_esami()
    insegnamenti = loader.carica_insegnamenti()

    if loader.errors:
        print(f"Attenzione: {len(loader.errors)} errori durante il
caricamento")
        # Salva log errori

    # Analizza
    print("Analisi in corso...")
    analyzer = Analyzer(studenti, esami, insegnamenti)

    # Genera report
    print("Generazione report...")
    reporter = ReportGenerator(output_dir)

    # TODO: Generare tutti i report richiesti

    print("Elaborazione completata!")

if __name__ == "__main__":
    main()
```

Criteri di valutazione progetto finale (totale 50 punti):

1. **Architettura e design** (10 punti):
   - Separazione responsabilità
   - Uso appropriato di classi e dataclass
   - Documentazione del codice
2. **Caricamento e validazione dati** (10 punti):
   - Parsing corretto di tutti i file
   - Validazione robusta
   - Gestione errori e logging
3. **Analisi dati** (15 punti):
   - Calcoli corretti per tutte le metriche
   - Efficienza algoritmica
   - Completezza delle analisi
4. **Generazione report** (10 punti):

    - Formato corretto CSV/JSON/Markdown
    - Completezza delle informazioni
    - Presentazione professionale

5. **Testing e robustezza** (5 punti):
    - Test unitari
    - Gestione casi limite
    - Documentazione utilizzo

Parte 5: Checklist e valutazione

Checklist competenze

Alla fine dell'esercitazione, dovreste essere in grado di:

- [ ] Utilizzare `csv.reader` e `csv.writer` per operazioni base
- [ ] Utilizzare `DictReader` e `DictWriter` per accesso strutturato
- [ ] Configurare dialect e formati CSV personalizzati
- [ ] Implementare conversione automatica dei tipi
- [ ] Gestire errori e validare dati CSV
- [ ] Processare file grandi con approccio streaming
- [ ] Implementare un parser CSV custom
- [ ] Integrare parsing CSV in progetti complessi
- [ ] Generare report in multipli formati
- [ ] Scrivere test per codice di parsing

Griglia di valutazione generale

Esercizio	Punti Max	Difficoltà	Peso
Es. 1 - Lettura base	10	★☆☆☆☆	5%
Es. 2 - DictReader	10	★★☆☆☆	8%
Es. 3 - Formati alternativi	10	★★☆☆☆	8%
Es. 4 - Validazione	12	★★★☆☆	12%
Es. 5 - Trasformazione	15	★★★☆☆	15%
Es. 6 - Streaming	15	★★★★☆	17%
Es. 7 - Parser custom	20	★★★★★	20%
Progetto finale	50	★★★★★	35%
Totale	**142**		**120%**

Nota: Il punteggio totale supera 100% per permettere recuperi e bonus

Modalità di consegna

1. **Esercizi singoli**:
    - File Python con nome `esercizio_N.py`
    - Commenti che spiegano l'approccio
    - Output di esempio come commento o file separato
2. **Progetto finale**:
    - Repository completo con struttura indicata
    - File README.md con istruzioni
    - Test funzionanti
    - File CSV di esempio inclusi
3. **Documentazione**:
    - Docstring per tutte le funzioni pubbliche
    - Commenti per logica complessa
    - Type hints dove appropriato

Risorse aggiuntive

Documentazione ufficiale

- Modulo csv Python
- RFC 4180
- PEP 305 - CSV File API

Best practices

- Usare sempre `with open()` per garantire chiusura file
- Specificare sempre `encoding='utf-8'` e `newline=''`
- Validare i dati prima di processarli
- Gestire gracefully errori di conversione
- Usare approccio streaming per file grandi
- Documentare assunzioni sul formato dei dati

Errori comuni da evitare

- Non gestire encoding dei caratteri
- Non specificare `newline=''` in apertura file
- Assumere che tutti i valori siano validi
- Caricare file interi in memoria
- Non validare numero di campi per record
- Ignorare errori di conversione tipo

Buon lavoro!

Per domande o chiarimenti, contattare il docente durante le ore di ricevimento o via email.