Working With CSV in Python

Introduction

Programs often need to work with **tabular data**.

For example:

Name	Age	Course
Hafsa	25	Python
Asim	26	JavaScript
Ali	24	Flutter

A common way to store this kind of data is a **CSV file**.

CSV stands for:

Comma Separated Values

A CSV file stores data in rows and columns using a simple text format.

For example:

```
Name,Age,Course
Hafsa,25,Python
Asim,26,JavaScript
Ali,24,Flutter
```

CSV is widely used for:

- spreadsheets
- student records
- sales data
- user lists
- reports
- datasets
- data analysis
- importing and exporting information between applications

Python has a built-in `csv` module that makes working with CSV files much easier.

---

1. What Is a CSV File?

A CSV file is a text file where values are separated by a delimiter, commonly a comma.

Example:

```
Name,Age,Course
Hafsa,25,Python
Asim,26,JavaScript
```

Think of it as a table.

```
CSV file

      Columns
        
Name | Age | Course
-------------------
Hafsa| 25  | Python
Asim | 26  | JavaScript
```

Each line usually represents one row.

Each comma separates one value from another.

---

2. Why Use CSV?

You could store the same information in a normal text file:

```
Hafsa | 25 | Python
Asim | 26 | JavaScript
```

But CSV gives the data a standard structure.

This makes it easier for programs such as:

- Python
- Excel
- Google Sheets
- databases
- data analysis tools

to exchange information.

For example:

Python program

CSV file

Excel / Google Sheets

CSV is therefore useful when different tools need to share tabular data.

---

3. CSV Is Not an Excel File

A CSV file is usually plain text.

For example:

students.csv

might contain:

```
Name,Age,Course
Hafsa,25,Python
Asim,26,JavaScript
```

An Excel workbook such as:

students.xlsx

is a different file format.

CSV generally does not store advanced spreadsheet features such as:

- formulas
- formatting
- multiple worksheets
- charts
- cell colors

CSV mainly stores structured rows and values.

---

4. Python's `csv` Module

Python includes a built-in module called:

`csv`

Import it using:

`import csv`

You do **not** need to install it separately.

Example:

`import csv`

This gives your program access to Python's CSV tools.

---

5. Reading a CSV File

Suppose we have:

`students.csv`

with:

```
Name,Age,Course
Hafsa,25,Python
Asim,26,JavaScript
Ali,24,Flutter
```

We can read it using:

```
import csv

with open("students.csv", "r", encoding="utf-8") as file:
    reader = csv.reader(file)

    for row in reader:
        print(row)
```

Output:

```
['Name', 'Age', 'Course']
['Hafsa', '25', 'Python']
['Asim', '26', 'JavaScript']
['Ali', '24', 'Flutter']
```

Each row is returned as a list.

---

6. Understanding `csv.reader()`

This line:

```
reader = csv.reader(file)
```

creates a CSV reader.

Then:

```
for row in reader:
    print(row)
```

goes through the CSV one row at a time.

The process is:

```
CSV file
   
csv.reader()
   
One row
   
List of values
```

For example:

```
Hafsa,25,Python
```

becomes:

```
["Hafsa", "25", "Python"]
```

---

7. Accessing Individual Values

Because each row is a list, you can use indexing.

```
import csv

with open("students.csv", "r", encoding="utf-8") as file:
    reader = csv.reader(file)

    for row in reader:
        print(row[0])
```

Output:

```
Name
Hafsa
Asim
Ali
```

`row[0]` means the first column.

Similarly:

`row[1]`

means the second column.

And:

`row[2]`

means the third column.

---

8. Skipping the Header

Many CSV files have a header row:

```
Name,Age,Course
```

Sometimes you want to process only the actual data.

You can use:

```
import csv

with open("students.csv", "r", encoding="utf-8") as file:
    reader = csv.reader(file)

    next(reader)

    for row in reader:
        print(row)
```

Output:

```
['Hafsa', '25', 'Python']
['Asim', '26', 'JavaScript']
['Ali', '24', 'Flutter']
```

`next(reader)` moves past the first row.

---

9. CSV Values Are Strings

This is important.

Suppose the CSV contains:

```
Hafsa,25,Python
```

When Python reads it:

```
["Hafsa", "25", "Python"]
```

The `25` is a string:

```
"25"
```

not an integer:

```
25
```

If you need to perform calculations, convert it:

```
age = int(row[1])
```

Example:

```
import csv

with open("students.csv", "r", encoding="utf-8") as file:
    reader = csv.reader(file)

    next(reader)

    for row in reader:
        age = int(row[1])
```

```
    print(age + 1)
```

---

10. Writing to a CSV File

Python can also create CSV files.

Use:

```
csv.writer()
```

Example:

```
import csv

with open("students.csv", "w", newline="", encoding="utf-8") as file:
    writer = csv.writer(file)

    writer.writerow(["Name", "Age", "Course"])
    writer.writerow(["Hafsa", 25, "Python"])
    writer.writerow(["Asim", 26, "JavaScript"])
```

The resulting file:

```
Name,Age,Course
Hafsa,25,Python
Asim,26,JavaScript
```

---

11. Understanding `writerow()`

This method:

`writer.writerow(...)`

writes one row.

Example:

```
writer.writerow(["Hafsa", 25, "Python"])
```

creates:

```
Hafsa,25,Python
```

You can call it repeatedly:

```
writer.writerow(["Hafsa", 25, "Python"])
writer.writerow(["Asim", 26, "JavaScript"])
writer.writerow(["Ali", 24, "Flutter"])
```

---

12. Writing Multiple Rows With `writerows()`

If you already have multiple rows, use:

`writer.writerows(...)`

Example:

```
import csv

students = [
    ["Name", "Age", "Course"],
    ["Hafsa", 25, "Python"],
```

```
    ["Asim", 26, "JavaScript"],
    ["Ali", 24, "Flutter"]
]

with open("students.csv", "w", newline="", encoding="utf-8") as file:
    writer = csv.writer(file)
    writer.writerows(students)
```

This creates the complete CSV.

The difference is:

```
writerow()  → one row
writerows() → multiple rows
```

---

13. Why `newline=""` Is Used

When writing CSV files, you will often see:

```
newline=""
```

Example:

```
with open("students.csv", "w", newline="", encoding="utf-8") as file:
```

This helps Python's CSV module handle newline characters correctly across platforms and helps avoid unwanted blank lines in some environments.

A common CSV-writing pattern is therefore:

```
with open("data.csv", "w", newline="", encoding="utf-8") as file:
```

```
    writer = csv.writer(file)
```

---

14. Using `csv.DictReader`

Sometimes lists are not the easiest way to work with CSV data.

Consider:

```
Name,Age,Course
Hafsa,25,Python
Asim,26,JavaScript
```

You can use:

```
csv.DictReader
```

Example:

```
import csv

with open("students.csv", "r", encoding="utf-8") as file:
    reader = csv.DictReader(file)

    for row in reader:
        print(row["Name"])
        print(row["Course"])
```

Output:

```
Hafsa
Python
```

```
Asim
JavaScript
```

This can be easier to understand because you access columns by their names.

---

15. Understanding `DictReader`

Instead of:

```
row[0]
row[1]
row[2]
```

you can use:

```
row["Name"]
row["Age"]
row["Course"]
```

Conceptually:

```
CSV header
    
Dictionary keys
    
Values
```

A row behaves approximately like:

```
{
   "Name": "Hafsa",
```

```
    "Age": "25",
    "Course": "Python"
}
```

This makes code more readable.

---

16. Writing Dictionaries With `DictWriter`

Python also provides:

`csv.DictWriter`

Example:

```
import csv

fieldnames = ["Name", "Age", "Course"]

students = [
    {
        "Name": "Hafsa",
        "Age": 25,
        "Course": "Python"
    },
    {
        "Name": "Asim",
        "Age": 26,
        "Course": "JavaScript"
    }
]

with open("students.csv", "w", newline="", encoding="utf-8") as file:
```

```
    writer = csv.DictWriter(file, fieldnames=fieldnames)

    writer.writeheader()
    writer.writerows(students)
```

The CSV becomes:

```
Name,Age,Course
Hafsa,25,Python
Asim,26,JavaScript
```

---

17. Understanding `writeheader()`

With `DictWriter`, you can write the column names using:

```
writer.writeheader()
```

For:

```
fieldnames = ["Name", "Age", "Course"]
```

it creates:

```
Name,Age,Course
```

Then:

```
writer.writerows(students)
```

writes the actual records.

---

18. Reading CSV With Dictionaries

This is often convenient for real-world applications.

```
import csv

with open("resources.csv", "r", encoding="utf-8") as file:
    reader = csv.DictReader(file)

    for resource in reader:
        print(
            resource["Title"],
            resource["Category"]
        )
```

Suppose the CSV contains:

```
Title,Category
Python Functions,Python
Working With CSV,Python
Git Basics,Developer Tools
```

The program can access:

```
resource["Title"]
```

and:

```
resource["Category"]
```

without remembering column numbers.

---

19. CSV Delimiters

A comma is the default delimiter.

For example:

```
Name,Age,Course
```

But CSV-like files can use other delimiters.

For example:

```
Name;Age;Course
Hafsa;25;Python
```

You can specify the delimiter:

```
import csv

with open("students.csv", "r", encoding="utf-8") as file:
    reader = csv.reader(file, delimiter=";")

    for row in reader:
        print(row)
```

The important idea is:

```
delimiter = character separating values
```

Common delimiters include:

```
,
```

;
\t

---

20. CSV Values Containing Commas

What if a value itself contains a comma?

For example:

```
Hafsa,"Gujranwala, Pakistan",Python
```

The CSV module understands that the comma inside quotes belongs to the value.

Example:

```
import csv

with open("students.csv", "r", encoding="utf-8") as file:
    reader = csv.reader(file)

    for row in reader:
        print(row)
```

Python can correctly interpret:

```
["Hafsa", "Gujranwala, Pakistan", "Python"]
```

This is one reason to use Python's `csv` module instead of manually splitting lines with:

```
line.split(",")
```

---

21. Why Not Just Use `split(",")`?

You might be tempted to write:

```
line.split(",")
```

For very simple data, it may appear to work.

But CSV has rules for:

- quoted values
- commas inside values
- different delimiters
- escaped quotes
- newline characters inside fields

The `csv` module handles these cases.

Therefore:

`Simple string splitting`

`Not a complete CSV parser`

`csv module`

`Designed for CSV rules`

Use the standard library instead of reinventing CSV parsing.

---

22. Filtering CSV Data

Once you can read CSV data, you can filter it.

Suppose:

```
Name,Age,Course
Hafsa,25,Python
Asim,26,JavaScript
Ali,24,Python
```

We want only Python students.

```
import csv

with open("students.csv", "r", encoding="utf-8") as file:
    reader = csv.DictReader(file)

    for row in reader:
        if row["Course"] == "Python":
            print(row["Name"])
```

Output:

```
Hafsa
Ali
```

This is where CSV starts becoming useful for actual data processing.

---

23. Calculating From CSV Data

Suppose we have:

```
Name,Marks
Hafsa,85
Asim,90
Ali,75
```

We can calculate an average:

```
import csv

total = 0
count = 0

with open("marks.csv", "r", encoding="utf-8") as file:
    reader = csv.DictReader(file)

    for row in reader:
        total += int(row["Marks"])
        count += 1

average = total / count

print("Average:", average)
```

The important pattern is:

```
Read CSV
   
Convert values
   
Process data
   
Calculate result
```

---

24. Creating a CSV From a List of Dictionaries

Suppose your Python program already has structured data:

```
resources = [
    {
        "title": "Python Functions",
        "category": "Python"
    },
    {
        "title": "Working With CSV",
        "category": "Python"
    },
    {
        "title": "Git Basics",
        "category": "Developer Tools"
    }
]
```

You can export it:

```
import csv

fieldnames = ["title", "category"]

with open("resources.csv", "w", newline="", encoding="utf-8") as file:
    writer = csv.DictWriter(file, fieldnames=fieldnames)

    writer.writeheader()
    writer.writerows(resources)
```

This produces:

```
title,category
Python Functions,Python
Working With CSV,Python
Git Basics,Developer Tools
```

This pattern is especially useful when exporting application data.

---

25. CSV Workflow in a Real Application

Imagine a student management application.

The workflow could be:

```
User enters student
        
Python receives information
        
Application validates it
        
Data is stored in CSV
        
Later, program reads CSV
        
Data is filtered/calculated
        
Report is generated
```

CSV can therefore act as a simple storage and exchange format for small applications.

For larger applications, databases are usually more appropriate.

---

26. CSV and haas.dev

Imagine haas.dev has resource metadata stored in Python:

```
resources = [
    {
        "title": "Python Functions",
        "category": "Python",
        "level": "Beginner"
    },
    {
        "title": "Working With CSV",
        "category": "Python",
        "level": "Beginner"
    }
]
```

You could export the information:

```
import csv

fieldnames = ["title", "category", "level"]

with open("resources.csv", "w", newline="", encoding="utf-8") as file:
    writer = csv.DictWriter(file, fieldnames=fieldnames)

    writer.writeheader()
    writer.writerows(resources)
```

Now the resource information can be opened in spreadsheet software or processed by another program.

This demonstrates an important software concept:

Application data
    ↓
Export
    ↓
Standard format
    ↓
Other tools can use it

---

27. CSV vs Normal Text Files

Feature	Text File	CSV
Stores plain text	Yes	Yes
Rows and columns	Manually structured	Naturally structured
Spreadsheet friendly	Limited	Yes
Built-in Python module	General file methods	`csv`
Good for tabular data	Sometimes	Yes
Supports structured columns	Not automatically	Yes

Use a normal text file for things like:

A paragraph
A note
A log message

Use CSV for things like:

```
Name,Age,Course
Hafsa,25,Python
```

---

28. CSV vs JSON

CSV is excellent for tabular data:

```
Name,Age,Course
Hafsa,25,Python
```

JSON is better when data has nested or more complex structures:

```
{
  "name": "Hafsa",
  "skills": [
    "Python",
    "JavaScript"
  ]
}
```

Think:

CSV
→
Rows + columns

JSON
→
Objects + nested data

Both are common data exchange formats.

---

29. Common Mistakes

Mistake 1: Forgetting to import `csv`

Incorrect:

```
reader = csv.reader(file)
```

Correct:

```
import csv
```

Mistake 2: Treating CSV numbers as integers

CSV values are read as strings.

```
age = row["Age"]
```

gives something like:

```
"25"
```

Convert when needed:

```
age = int(row["Age"])
```

Mistake 3: Forgetting the header

If using `csv.reader`, remember that the first row may be column names.

```
next(reader)
```

can skip it.

With `DictReader`, the first row is normally used as field names automatically.

Mistake 4: Using `split(",")`

Avoid manually parsing CSV when the data may contain quoted values.

Use:

```
csv.reader()
```

instead.

Mistake 5: Forgetting `newline=""` when writing

Use:

```
with open(
    "data.csv",
```

```
    "w",
    newline="",
    encoding="utf-8"
) as file:
```

for standard CSV writing.

Mistake 6: Using the wrong column name

If the CSV contains:

```
Name,Age,Course
```

this:

```
row["course"]
```

does not match:

```
row["Course"]
```

Column names must match.

---

30. Best Practices

Use the built-in CSV module

```
import csv
```

Use `with open()`

```
with open("data.csv", "r", encoding="utf-8") as file:
    ...
```

Use UTF-8 for text

```
encoding="utf-8"
```

Use `newline=""` when writing CSV

```
newline=""
```

Use `DictReader` for readable column access

```
reader = csv.DictReader(file)
```

Use `DictWriter` when exporting dictionaries

```
writer = csv.DictWriter(
    file,
    fieldnames=fieldnames
)
```

Convert values when calculations require numbers

```
score = int(row["Score"])
```

Validate important data

Do not assume every CSV row is perfectly formatted.

---

31. A Practical Problem-Solving Framework

When working with CSV, think in five steps.

Step 1: Understand the structure

Ask:

```
What are the columns?
What does each row represent?
```

Step 2: Choose the reader

Simple rows:

```
csv.reader()
```

Named columns:

```
csv.DictReader()
```

Step 3: Read the data

```
for row in reader:
    ...
```

Step 4: Process the data

You might:

```
filter
search
calculate
sort
validate
transform
```

Step 5: Export results

Use:

`csv.writer()`

or:

`csv.DictWriter()`

The overall pattern becomes:

```
CSV
 
Read
 
Understand
 
Process
 
Write
 
New CSV
```

---

32. Mini Project: Student CSV Manager

Create a program that stores student information.

```
import csv

students = [
    {
        "name": "Hafsa",
        "age": 25,
```

```
        "course": "Python"
    },
    {
        "name": "Asim",
        "age": 26,
        "course": "JavaScript"
    },
    {
        "name": "Ali",
        "age": 24,
        "course": "Python"
    }
]

fieldnames = ["name", "age", "course"]

with open("students.csv", "w", newline="", encoding="utf-8") as file:
    writer = csv.DictWriter(file, fieldnames=fieldnames)

    writer.writeheader()
    writer.writerows(students)

print("Students saved.")
```

Then read them:

```
import csv

with open("students.csv", "r", encoding="utf-8") as file:
    reader = csv.DictReader(file)

    for student in reader:
        print(
            student["name"],
```

```
        student["course"]
    )
```

---

33. Mini Project: Resource Exporter

Create a simple resource exporter for a learning platform.

```
import csv

resources = [
    {
        "title": "Python Functions",
        "category": "Python",
        "level": "Beginner"
    },
    {
        "title": "Reading Files",
        "category": "Python",
        "level": "Beginner"
    },
    {
        "title": "Working With CSV",
        "category": "Python",
        "level": "Beginner"
    }
]

fieldnames = ["title", "category", "level"]

with open("resources.csv", "w", newline="", encoding="utf-8") as file:
    writer = csv.DictWriter(file, fieldnames=fieldnames)
```

```
    writer.writeheader()
    writer.writerows(resources)

print("Resource CSV created.")
```

The resulting CSV can be opened in a spreadsheet application.

---

34. 5 Minute Challenge

Create a CSV file called:

`products.csv`

Your program should save:

```
Name,Price,Category
Keyboard,2500,Accessories
Mouse,1500,Accessories
Monitor,30000,Display
```

Then write another program that:

1. reads the CSV
2. prints each product
3. converts the prices to integers
4. calculates the total price
5. prints the most expensive product

Extra challenge

Filter the products so that only:

`Accessories`

are displayed.

---

35. Exercises

Exercise 1

Create a CSV containing:

```
Name
Age
City
```

for three people.

Exercise 2

Read the CSV using `csv.reader()`.

Exercise 3

Skip the header row.

Exercise 4

Read the same file using `csv.DictReader()`.

Exercise 5

Print only the names.

Exercise 6

Create a CSV of five students and their marks.

Exercise 7

Calculate the average marks.

Exercise 8

Print only students whose marks are greater than 80.

Exercise 9

Create a list of dictionaries representing haas.dev resources and export it using `DictWriter`.

Exercise 10

Read the generated resource CSV and print only resources from the Python category.

---

36. Mini Quiz

Question 1

What does CSV stand for?

A. Code Storage Value

B. Comma Separated Values

C. Computer Stored Variables

D. Common String Values

Answer: B

Question 2

Which Python module is used for CSV files?

A. `data`

B. `table`

C. `csv`

D. `file`

Answer: C

Question 3

Which function reads CSV rows as lists?

A. `csv.reader()`

B. `csv.read()`

C. `csv.open()`

D. `csv.rows()`

Answer: A

Question 4

Which class lets you access CSV columns by their names?

A. `csv.Reader`

B. `csv.DictReader`

C. `csv.ColumnReader`

D. `csv.NamedReader`

Answer: B

Question 5

Which method writes one CSV row?

A. `write()`

B. `writerow()`

C. `writerows()`

D. `writecsv()`

Answer: B

Question 6

Which method writes multiple CSV rows?

A. `writerows()`

B. `manyrows()`

C. `writeall()`

D. `rows()`

Answer: A

---

37. Interview Questions

1. What is CSV?

CSV is a simple text-based format used to store tabular data as rows and columns.

2. How do you read a CSV file in Python?

Use the built-in `csv` module and `csv.reader()` or `csv.DictReader()`.

3. What is the difference between `reader()` and `DictReader()`?

`reader()` returns rows as lists, while `DictReader()` allows values to be accessed using column names.

4. What is `writerow()`?

It writes one row to a CSV file.

5. What is `writerows()`?

It writes multiple rows.

6. What is `DictWriter`?

It writes dictionaries to CSV using specified field names.

7. Why are CSV numbers usually converted using `int()` or `float()`?

CSV values are read as strings, so numeric values must be converted before numerical calculations.

8. Why should you use Python's `csv` module instead of `split(",")`?

The CSV module understands CSV-specific rules such as quoted fields and delimiters.

9. Why is `newline=""` commonly used when writing CSV files?

It allows the CSV module to handle newline characters correctly.

10. When is CSV a poor choice?

For complex nested data, relational queries, concurrent application storage, or large production systems, databases or other structured formats may be more appropriate.

---

38. Cheat Sheet

Import

```
import csv
```

Read rows

```
with open("data.csv", "r", encoding="utf-8") as file:
    reader = csv.reader(file)

    for row in reader:
        print(row)
```

Skip header

```
next(reader)
```

Read dictionaries

```
reader = csv.DictReader(file)

for row in reader:
    print(row["Name"])
```

Write one row

```
writer.writerow(["Hafsa", 25, "Python"])
```

Write multiple rows

```
writer.writerows(rows)
```

Write dictionaries

```
writer = csv.DictWriter(
    file,
    fieldnames=["Name", "Age"]
)

writer.writeheader()
writer.writerows(data)
```

Standard CSV writing pattern

```
with open(
    "data.csv",
    "w",
    newline="",
    encoding="utf-8"
) as file:
    writer = csv.writer(file)
```

Custom delimiter

```
reader = csv.reader(
    file,
    delimiter=";"
)
```

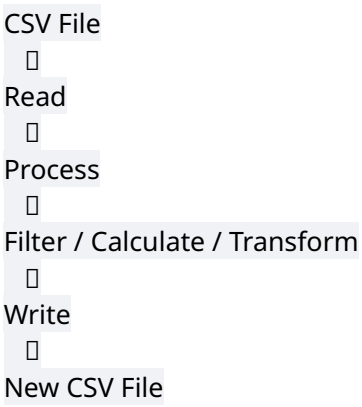
---

39. Key Takeaways

- CSV stands for **Comma Separated Values**.
- CSV is commonly used for tabular data.
- Python provides a built-in `csv` module.
- `csv.reader()` reads rows as lists.
- `csv.DictReader()` provides access using column names.

- `csv.writer()` writes CSV rows.
- `csv.DictWriter()` writes dictionaries.
- `writerow()` writes one row.
- `writerows()` writes multiple rows.
- `writeheader()` writes column names when using `DictWriter`.
- CSV values are normally read as strings.
- Convert numeric values before calculations.
- Use `newline=""` when writing CSV files.
- Use `encoding="utf-8"` for text handling.
- Avoid manually parsing CSV with `split(",")`.
- CSV is useful for data exchange, exports, reports, and small datasets.
- For complex nested data, JSON may be more appropriate.
- For larger applications requiring querying and reliable persistent storage, databases are generally more suitable.

The core CSV workflow is:



Once you understand CSV, you can start moving from simple file storage toward **real data processing and data exchange**.

Visit haas.dev for more resources and guides.

https://dev-roast-app.vercel.app

haas.dev

https://dev-roast-app.vercel.app/resources