Python Reading Files

Programs often need to work with information stored outside the Python code itself.

For example:

- User data
- Configuration files
- Logs
- Text documents
- CSV data
- Saved application information

Python allows us to **read files** and use their contents inside our programs.

The basic process is:

```
Open file
   
Read data
   
Process data
   
Close file
```

Python provides the built in `open()` function for working with files.

---

1. Why Read Files?

Imagine a program that needs a list of resources.

Instead of writing everything directly in Python:

```
resources = ["Python", "JavaScript", "Django"]
```

we could store the data in a file:

```
resources.txt
```

Then Python can read that file.

This allows data to exist separately from the program.

---

2. The `open()` Function

The basic syntax is:

```
open(filename, mode)
```

For reading a text file:

```
file = open("data.txt", "r")
```

Here:

- `"data.txt"` → file name
- `"r"` → read mode

The `"r"` means:

> Open the file for reading.

---

3. Reading the Entire File

The simplest way to read a file is with `.read()`.

Suppose `data.txt` contains:

```
Python
JavaScript
Django
```

Python:

```
file = open("data.txt", "r")

content = file.read()

print(content)

file.close()
```

Output:

```
Python
JavaScript
Django
```

---

4. Understanding `.read()`

The `.read()` method reads the file's contents and returns them as a string.

```
file = open("data.txt", "r")

content = file.read()

print(type(content))

file.close()
```

Output:

```
<class 'str'>
```

So:

```
file.read()
```

returns a string.

---

5. Closing the File

After working with a file, we should close it.

```
file = open("data.txt", "r")

content = file.read()

file.close()
```

Why?

Opening a file uses an operating system resource.

Closing it tells Python:

> I'm finished using this file.

---

6. What Happens If You Don't Close It?

For a tiny script, you may not immediately notice a problem.

But in larger applications, leaving files open can cause:

- unnecessary resource usage
- too many open files
- locked resources
- unexpected behavior

So file resources should be handled properly.

---

7. The Better Way: `with open()`

Python provides a safer and cleaner pattern:

```
with open("data.txt", "r") as file:
    content = file.read()

print(content)
```

The `with` statement automatically handles closing the file.

You don't need:

```
file.close()
```

---

8. Why Use `with`?

Compare:

Manual closing

```
file = open("data.txt", "r")

content = file.read()

file.close()
```

Context manager

```
with open("data.txt", "r") as file:
    content = file.read()
```

The second approach is generally preferred.

The file is automatically cleaned up when the `with` block finishes.

---

9. Reading Line by Line

Sometimes the file is large.

Instead of reading the entire file at once, we can process it line by line.

```
with open("data.txt", "r") as file:

    for line in file:
        print(line)
```

If the file contains:

```
Python
JavaScript
Django
```

the loop processes:

```
Python
JavaScript
Django
```

one line at a time.

This can be more memory efficient for large files.

---

10. Why Read Line by Line?

Imagine a log file containing:

```
10,000,000 lines
```

Doing:

```
content = file.read()
```

loads the entire file into memory.

Instead:

```
for line in file:
    process(line)
```

allows the program to process the file gradually.

Mental model:

```
read()
↓
Entire file → memory

for line in file
↓
One line → process → next line
```

---

11. Using `.readline()`

The `.readline()` method reads one line at a time.

```
with open("data.txt", "r") as file:

    line = file.readline()

    print(line)
```

If the file is:

```
Python
JavaScript
Django
```

the first call reads:

```
Python
```

---

12. Reading Multiple Lines With `readline()`

You can call it repeatedly:

```
with open("data.txt", "r") as file:

    line1 = file.readline()
    line2 = file.readline()
    line3 = file.readline()

    print(line1)
    print(line2)
    print(line3)
```

Each call moves the file position forward.

---

13. The File Cursor

When Python reads a file, it keeps track of where it currently is.

This position is often called the **file cursor** or file pointer.

Imagine:

```
Python
JavaScript
Django
```

Initially:

```
▯
Python
JavaScript
Django
```

After reading the first line:

```
Python
      ▯
JavaScript
Django
```

The cursor moves forward as data is read.

---

14. Reading With `.read()`

Suppose:

```
with open("data.txt", "r") as file:

    print(file.read())
    print(file.read())
```

The second `read()` may return an empty string.

Why?

Because the cursor has already reached the end of the file.

---

15. Resetting the File Cursor With `seek()`

Python provides `.seek()` to move the cursor.

```
with open("data.txt", "r") as file:
    print(file.read())
    file.seek(0)
    print(file.read())
```

`seek(0)` moves the cursor back to the beginning.

---

16. Getting the Current Position With `tell()`

Use `.tell()` to find the current cursor position.

```
with open("data.txt", "r") as file:
    print(file.tell())
    file.read(5)
```

```
    print(file.tell())
```

The exact position depends on the file contents and encoding.

The important idea is:

```
tell() → Where am I?
seek() → Move me somewhere.
```

---

17. Reading a Specific Number of Characters

`.read()` can accept a number.

```
with open("data.txt", "r") as file:

    content = file.read(5)

    print(content)
```

If the file starts with:

```
Python programming
```

the result is approximately:

```
Pytho
```

The number represents how much data to read.

---

18. Reading All Lines With `.readlines()`

Python also provides:

`readlines()`

Example:

```
with open("data.txt", "r") as file:

    lines = file.readlines()

print(lines)
```

If the file contains:

```
Python
JavaScript
Django
```

you may get:

```
[
    "Python\n",
    "JavaScript\n",
    "Django"
]
```

So `.readlines()` returns a **list of strings**.

---

19. `.read()` vs `.readline()` vs `.readlines()`

Method	Returns	Typical use
`.read()`	Entire content as a string	Small/medium files
`.read(size)`	Up to a specified amount	Controlled reading
`.readline()`	One line	Read specific lines
`.readlines()`	List of lines	When you need all lines as a list
`for line in file`	One line at a time	Large files

---

20. Removing Newline Characters

When reading lines, you may see:

```
"Python\n"
```

The `\n` represents a newline.

You can remove surrounding whitespace:

```
with open("data.txt", "r") as file:

    for line in file:
        print(line.strip())
```

Now:

```
Python
JavaScript
Django
```

is printed without the extra newline behavior.

---

21. Be Careful With `.strip()`

`.strip()` removes whitespace from both ends.

Example:

```
text = "  Python  "

print(text.strip())
```

Output:

```
Python
```

When processing files, this can be useful.

But don't automatically use `.strip()` if leading or trailing whitespace is meaningful to your data.

---

22. Checking Whether a File Exists

If the file doesn't exist:

```
with open("missing.txt", "r") as file:
    content = file.read()
```

Python raises:

```
FileNotFoundError
```

We can handle it:

```
try:
    with open("missing.txt", "r") as file:
        content = file.read()

except FileNotFoundError:
    print("File does not exist")
```

---

23. Reading Files Safely

A practical pattern is:

```
try:
    with open("data.txt", "r") as file:
        content = file.read()

except FileNotFoundError:
    print("File not found")

else:
    print(content)
```

This combines concepts from exception handling:

try
 □
Attempt to open/read
 □
Error?
 □□□ Yes □ except
 □□□ No □ else

The with statement handles closing the file.

---

24. Reading Files With Encoding

Text files use character encodings.

A common choice is UTF-8:

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

Using an explicit encoding can make text handling more predictable, especially when files contain characters beyond basic English.

For example:

Python
café
پاکستان

---

25. Why Encoding Matters

Different encodings represent characters differently.

If Python uses an incompatible encoding, you may get:

`UnicodeDecodeError`

For many modern text files, UTF-8 is a good default:

`encoding="utf-8"`

---

26. Reading a File Into a List

Suppose:

```
Python
JavaScript
Flutter
React
```

You can create a clean list:

```
with open("technologies.txt", "r", encoding="utf-8") as file:
    technologies = [line.strip() for line in file]

print(technologies)
```

Output:

```
[
    "Python",
    "JavaScript",
```

```
    "Flutter",
    "React"
]
```

Now the file data can be processed like a normal Python list.

---

27. Searching a File

You can search through lines:

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

    for line in file:
        if "Python" in line:
            print("Python found")
```

This is useful for:

- logs
- configuration files
- text data
- simple search tools

---

28. Counting Lines

```
count = 0

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

    for line in file:
```

```
        count += 1

print(f"Total lines: {count}")
```

This processes the file one line at a time.

---

29. Counting Words

A simple word counter:

```
word_count = 0

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

    for line in file:
        words = line.split()
        word_count += len(words)

print(f"Total words: {word_count}")
```

This demonstrates an important workflow:

```
File
 
Line
 
Words
 
Count
```

---

30. Reading a Configuration File

Imagine:

```
app_name=haas.dev
language=Python
environment=development
```

Python can read it:

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

    for line in file:
        key, value = line.strip().split("=")
        print(key, value)
```

This turns raw file content into structured information.

---

31. Reading Files From Different Directories

You can provide a path:

```
with open("data/users.txt", "r", encoding="utf-8") as file:
    content = file.read()
```

Or:

```
with open("data/2026/users.txt", "r", encoding="utf-8") as file:
    content = file.read()
```

Python interprets the path relative to the program's current working directory unless you provide an absolute path.

---

32. Relative vs Absolute Paths

Relative path

`open("data.txt")`

Means:

> Find `data.txt` relative to the current working directory.

Relative folder path

`open("data/users.txt")`

Means:

> Look inside the `data` directory.

Absolute path

`open("C:/projects/app/data.txt")`

This points to a specific location on the computer.

In real projects, relative paths are often easier to manage across different environments.

---

33. Using `pathlib`

Modern Python applications often use `pathlib` for paths.

```
from pathlib import Path

file_path = Path("data.txt")

with file_path.open("r", encoding="utf-8") as file:
    content = file.read()

print(content)
```

You can also use:

```
from pathlib import Path

content = Path("data.txt").read_text(encoding="utf-8")

print(content)
```

For simple text files, this can be very convenient.

---

34. Reading a File With `pathlib`

Example:

```
from pathlib import Path

file_path = Path("resources.txt")

if file_path.exists():
    content = file_path.read_text(encoding="utf-8")
```

```
    print(content)
else:
    print("File not found")
```

This separates:

```
Path handling
+
File reading
```

cleanly.

---

35. Reading Large Files

For a large file, avoid blindly doing:

```
content = file.read()
```

if the entire file could consume a lot of memory.

Prefer:

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

    for line in file:
        process(line)
```

This lets Python process the file incrementally.

---

36. Example: Processing a Log File

Imagine:

```
INFO User logged in
ERROR Database connection failed
INFO User logged out
ERROR File not found
```

We can find errors:

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

    for line in file:
        if "ERROR" in line:
            print(line.strip())
```

Output:

```
ERROR Database connection failed
ERROR File not found
```

This is a simple example of how real applications process logs.

---

37. Example: Reading haas.dev Resources

Imagine a file containing:

```
Python Basics
Python Functions
Python Modules
```

```
Python Exceptions
```

We can load the resource titles:

```
with open("resources.txt", "r", encoding="utf-8") as file:
    resources = [line.strip() for line in file]

for resource in resources:
    print(resource)
```

Output:

```
Python Basics
Python Functions
Python Modules
Python Exceptions
```

The important idea is:

> A file can act as a simple external source of application data.

---

38. File Reading and Data Processing

Reading a file is usually only the first step.

A real workflow might be:

```
Open file
  
```

Read data
  ↓
Clean data
  ↓
Convert data
  ↓
Validate data
  ↓
Process data
  ↓
Store/use result

For example:

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

    users = []

    for line in file:
        username = line.strip()

        if username:
            users.append(username)

print(users)
```

---

39. Common Mistakes

Mistake 1: Forgetting the file mode

You can write:

`open("data.txt", "r")`

The `"r"` clearly communicates that the file is being opened for reading.

Mistake 2: Forgetting to close manually opened files

Avoid:

```
file = open("data.txt", "r")
content = file.read()
```

Prefer:

```
with open("data.txt", "r") as file:
    content = file.read()
```

Mistake 3: Reading huge files into memory

Avoid:

```
content = file.read()
```

for very large files when you only need to process them incrementally.

Prefer:

```
for line in file:
    process(line)
```

Mistake 4: Ignoring encoding

For text files, explicitly using UTF-8 is often clearer:

```
open("data.txt", "r", encoding="utf-8")
```

Mistake 5: Assuming the file always exists

This can crash:

```
with open("users.txt", "r") as file:
    ...
```

If the file might not exist, handle:

```
FileNotFoundError
```

---

40. Best Practices

1. Prefer `with open()`

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

2. Use UTF-8 for normal text files

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

3. Process large files incrementally

```
for line in file:
```

```
    ...
```

4. Handle expected file errors

```
try:
    ...
except FileNotFoundError:
    ...
```

5. Keep file handling separate from business logic

Instead of mixing everything:

```
open file
parse data
validate user
calculate result
display result
```

consider separating responsibilities into functions.

---

41. A Better Program Structure

Instead of:

```
with open("users.txt", "r") as file:
    users = []

    for line in file:
        username = line.strip()

        if username:
```

```
            users.append(username)

    for user in users:
        print(user)
```

we can separate reading:

```
def read_users(filename):

    with open(filename, "r", encoding="utf-8") as file:
        return [
            line.strip()
            for line in file
            if line.strip()
        ]
```

Then:

```
users = read_users("users.txt")

for user in users:
    print(user)
```

Now the function has one clear responsibility:

> Read users from a file.

---

42. File Reading With Error Handling

A reusable version:

```
def read_users(filename):

    try:
        with open(filename, "r", encoding="utf-8") as file:
            return [
                line.strip()
                for line in file
                if line.strip()
            ]

    except FileNotFoundError:
        raise FileNotFoundError(
            f"Could not find file: {filename}"
        )
```

The function reads the file and provides a more informative error.

---

43. Reading Empty Files

An empty file is not necessarily an error.

```
with open("empty.txt", "r", encoding="utf-8") as file:
    content = file.read()

print(content)
```

The result is:

```
""
```

An empty string represents no content.

Your application should decide whether an empty file is acceptable.

---

44. Reading Files With `readline()` and EOF

When there are no more lines:

```
line = file.readline()
```

returns:

```
""
```

This represents the end of the file.

You can use:

```
while True:
    line = file.readline()

    if line == "":
        break

    print(line)
```

However, for ordinary line-by-line reading, this is usually cleaner:

```
for line in file:
    print(line)
```

---

45. Reading Binary Files

Not every file is text.

Images, PDFs, audio, and many other files contain binary data.

For binary reading, use:

```
with open("image.jpg", "rb") as file:
    data = file.read()
```

Here:

```
r  → text read mode
rb → binary read mode
```

The result of binary reading is usually a `bytes` object.

```
print(type(data))
```

Output:

```
<class 'bytes'>
```

Binary files are covered more deeply when working with file types and binary data.

---

46. Text vs Binary Reading

Mode	Purpose
"r"	Read text
"rb"	Read binary data

Examples:

`open("notes.txt", "r", encoding="utf-8")`

`open("photo.jpg", "rb")`

Use text mode for text and binary mode for binary files.

---

47. Mini Project: Text File Analyzer

Build a program that reads a text file and calculates:

- number of lines
- number of words
- number of characters

```
def analyze_file(filename):

    lines = 0
    words = 0
    characters = 0
```

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

        for line in file:
            lines += 1
            words += len(line.split())
            characters += len(line)

    return lines, words, characters
```

Use:

```
lines, words, characters = analyze_file("data.txt")

print(f"Lines: {lines}")
print(f"Words: {words}")
print(f"Characters: {characters}")
```

This combines:

```
File reading
+
Loops
+
Strings
+
Lists/variables
+
Functions
```

---

48. 5 Minute Challenge

Create:

```
def find_keyword(filename, keyword):
    ...
```

Requirements:

1. Open the file safely.
2. Read it line by line.
3. Find lines containing the keyword.
4. Print matching lines.
5. Handle `FileNotFoundError`.

Example:

```
find_keyword("app.log", "ERROR")
```

Expected behavior:

```
ERROR Database connection failed
ERROR File not found
```

---

49. Exercises

Exercise 1

Create a file called:

```
names.txt
```

Add five names.

Write Python code that reads and prints every name.

Exercise 2

Read a file and count how many lines it contains.

Exercise 3

Read a file and count the total number of words.

Exercise 4

Read a file and print only lines containing:

`Python`

Exercise 5

Create:

```
def read_resource_titles(filename):
    ...
```

Return all non-empty lines as a list.

Exercise 6

Modify the function so that it handles a missing file.

Exercise 7

Read a large file line by line and count how many lines contain:

`ERROR`

---

50. Mini Quiz

1. Which function opens a file?

A. `read()`
B. `open()`
C. `file()`
D. `load()`

Answer: B

---

2. Which mode is used to read a text file?

A. `"w"`
B. `"a"`
C. `"r"`
D. `"x"`

Answer: C

---

3. What does `.read()` return?

A. A list
B. A dictionary
C. A string containing the file contents
D. An integer

Answer: C

---

4. Which method reads one line?

A. `.readline()`
B. `.readall()`
C. `.line()`
D. `.getline()`

Answer: A

---

5. Which approach automatically closes the file?

A. `if open()`
B. `with open()`
C. `try open()`
D. `for open()`

Answer: B

---

6. Which exception is raised when a requested file doesn't exist?

A. `ValueError`
B. `KeyError`
C. `FileNotFoundError`
D. `NameError`

Answer: C

---

51. Interview Questions

Beginner

1. How do you open a file in Python?
2. What does `"r"` mean?
3. What does `.read()` do?
4. What does `.readline()` do?
5. What does `.readlines()` return?
6. Why should files be closed?
7. What is the purpose of `with open()`?

Intermediate

8. What is the difference between `.read()`, `.readline()`, and `.readlines()`?
9. Why might you read a large file line by line?
10. What is a file cursor?
11. What do `seek()` and `tell()` do?
12. Why is specifying `encoding="utf-8"` useful?
13. How do you handle `FileNotFoundError`?
14. What is the difference between text mode and binary mode?

Practical

15. How would you count errors in a large log file?
16. How would you read a file into a list?
17. How would you search for a keyword in a file?
18. How would you design a reusable file-reading function?
19. How would you safely process a file that may not exist?
20. How would you read a large file without loading all of it into memory?

---

52. File Reading Cheat Sheet

Open and read

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

Read one line

```
line = file.readline()
```

Read all lines

```
lines = file.readlines()
```

Read line by line

```
for line in file:
    print(line)
```

Remove newline

```
line.strip()
```

Move cursor

```
file.seek(0)
```

Get cursor position

```
file.tell()
```

Handle missing file

```
try:
    with open("data.txt", "r", encoding="utf-8") as file:
        content = file.read()

except FileNotFoundError:
    print("File not found")
```

Binary reading

```
with open("image.jpg", "rb") as file:
    data = file.read()
```

---

53. Key Takeaways

- Python uses `open()` to open files.
- `"r"` means read mode.
- `.read()` reads the file's content as a string.
- `.readline()` reads one line.
- `.readlines()` returns a list of lines.
- Iterating directly over a file is useful for line-by-line processing.
- `with open()` is the preferred way to manage files because it automatically closes the file.
- `seek()` moves the file cursor.
- `tell()` reports the current cursor position.
- UTF-8 is a common encoding for text files.

- `FileNotFoundError` should be handled when a file may not exist.
- Large files are often better processed incrementally rather than loaded completely into memory.
- Text files use modes such as `"r"`, while binary files use `"rb"`.
- File reading is usually the first step before cleaning, validating, searching, analyzing, or transforming data.

The core pattern to remember is:

```
try:
    with open("data.txt", "r", encoding="utf-8") as file:
        for line in file:
            process(line)

except FileNotFoundError:
    print("File not found")
```

The important mental model is:

```
File
 
Open
 
Read
 
Process
 
Close automatically
```

Once you understand file reading, you can start building programs that work with **real data stored outside your Python code**, which is an important step toward practical Python applications.

Visit haas.dev for more resources and guides.

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

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