Module-3

Dictionaries: Dictionary operations, dictionary methods, aliasing and copying.

Numpy: About, Shape, Slicing, masking, Broadcasting, dtype**.**

Files: About files, writing our first file, reading a file line-at-a-time, turning a file into a list of lines, Reading the whole file at once, working with binary files, Directories, fetching something from the Web.
Chapter: 5.4, 6.1-6.5, 7.1-7.8

3.1 Dictionaries – Simplified Explanation

So far, we have studied **strings, lists, and tuples**, which are called **sequence types**.
Sequence types store items in a specific order and use **integer indices** (0, 1, 2, …) to access

elements. A **dictionary** is a different kind of compound data type in Python.

What Is a Dictionary?

A **dictionary** is Python's built-in **mapping type**.

- It maps **keys → values**
- Keys must be **immutable** (e.g., strings, numbers, tuples)
- Values can be **any type** (numbers, strings, lists, etc.)
- Dictionaries are also called **associative arrays** in other languages

Creating a Dictionary

1. Creating an Empty Dictionary and Adding Items

english_spanish = {}

english_spanish["one"] =

"uno" english_spanish["two"]

= "dos" Here:

- "one" and "two" are **keys**
- "uno" and "dos" are

values Printing the dictionary:
print(english_spanish)

Possible output:

{'two': 'dos', 'one': 'uno'}

The order may not be the same as insertion order.

Dictionary Structure

Each item in a dictionary is written as:

key : value

Items are separated by commas and enclosed in { }.

Example:

{"one": "uno", "two": "dos"}

Why Order Looks Unpredictable (Hashing)

Dictionaries use a technique called **hashing** to store data.

- Hashing allows **very fast access**

- The internal order of items is **not meant to be relied upon**
- For learning purposes, treat dictionary order as **unpredictable**

Why Use Dictionaries Instead of Lists of Tuples?

Consider two ways to store fruit quantities:

Using a Dictionary

{'apples': 430, 'bananas': 312, 'oranges': 525, 'pears': 217}

Using a List of Tuples

[('apples', 430), ('bananas', 312), ('oranges', 525), ('pears', 217)]

Key Difference

- **Dictionary** → very fast lookup using keys
- **List of tuples** → slow lookup (must search item by item)

If the key is not found in a list, Python must scan the entire list.
Dictionaries avoid this problem using hashing.

Creating a Dictionary in One Step

```
english_spanish = {
    "one": "uno",
    "two": "dos",
    "three": "tres"
}
```

- Order does not matter
- Values are accessed using **keys**, not

positions **Accessing Values Using Keys**

print(english_spanish["two"])

Output:

dos

Her

e:

- "two" is the key
- "dos" is the value returned

Important Difference from Sequences

Feature	**Lists / Tuples / Strings**	**Dictionaries**
Access method	Index (0,1,2…)	Key
Ordered	Yes	No (conceptually)
Slicing	Yes	No
Mapping Dictionaries:	No	Yes

- Are **not sequences**
- Cannot be indexed or sliced
- Use **keys instead of indices**

3.1.1 Dictionary Operations

Python dictionaries allow us to **add, update, remove, and inspect** key–value pairs easily. Let's understand the most common operations using an example.

Example Dictionary

```
inventory = {
    "apples": 430,
    "bananas": 312,
```

```
    "oranges": 525,
    "pears": 217
}
```

This dictionary stores:

- **keys** → fruit names
- **values** → number of fruits in stock

Deleting a Key–Value Pair (del)

If all bananas are sold, we can remove bananas from the dictionary using the del statement: del inventory["bananas"]

print(inventory)

Output:

{'apples': 430, 'oranges': 525, 'pears': 217}

Important Note

- After deletion, trying to access inventory["bananas"] will cause a **KeyError**
- This is because the key no longer exists in the dictionary

Adding or Updating a Key–Value Pair

If we expect more bananas later, we can **add the key again** or **update its value**:

inventory["bananas"] = 0

print(inventory)

Output:

{'pears': 217, 'apples': 430, 'oranges': 525, 'bananas': 0}

- If the key already exists → value is **updated**
- If the key does not exist → a **new key–value pair** is created

Updating a Value Using Arithmetic

When a new shipment arrives:

inventory["bananas"] += 200

print(inventory)

Output:

{'pears': 217, 'apples': 430, 'oranges': 525, 'bananas': 200}

This works because:

- Dictionaries allow direct access to values using keys
- Values can be modified like normal variables

Finding the Size of a Dictionary (len)

The len() function returns the **number of key–value pairs** in a dictionary:

len(inventory)

Output:

4

This means the dictionary currently has **4**

keys. **Dictionary Operations**

Operation	Example	Purpose
Delete key	del inventory["bananas"]	Removes a key–value pair
Add key	inventory["bananas"] = 0	Adds a new entry
Update value	inventory["bananas"] += 200	Modifies existing value
Count items	len(inventory)	Number of key–value pairs

- del removes a key completely
- Assigning to a key adds or updates it
- Accessing a missing key causes an error
- Dictionary values are mutable
- len() returns number of key–value pairs
- Dictionaries are ideal for **inventory-type data**

3.1.2 Dictionary Methods

Python dictionaries provide several useful **built-in methods** that allow us to access and work with their contents efficiently. The most commonly used methods are:

- keys()
- values()
- items()

These methods return **view objects**, not lists.

Dictionary Views

A **view object** is a *lazy* object:

- It does not store a separate copy of the data

- It provides access to dictionary data when needed
- Similar to how range()

works You can:

- Iterate directly over a view
- Convert it into a list using list()

The keys() Method

The keys() method returns a view of all **keys** in the dictionary.

Example

```
for key in english_spanish.keys(): # Order is not guaranteed
    print("Got key", key, "which maps to value",
    english_spanish[key])
```

Possible output:

Got key three which maps to value

tres Got key two which maps to value

dos Got key one which maps to value

uno **Convert Keys to a List**

```
keys = list(english_spanish.keys())
print(keys)
```

Output:

['three', 'two', 'one']

Iterating Directly Over a Dictionary

Iterating over a dictionary **automatically iterates over its keys**, so keys() can be omitted: for key in english_spanish:

```
    print("Got key", key)
```

This is the **most common and Pythonic** way.

The values() Method

The values() method returns a view of all **values** in the dictionary.

Example

```
list(english_spanish.values())
```

Output:

['tres', 'dos', 'uno']

The items() Method

The items() method returns a view of **(key, value) tuples**.

Example

list(english_spanish.items())

Output:

[('three', 'tres'), ('two', 'dos'), ('one', 'uno')]

Looping with items() (Very Useful)

Tuples returned by items() allow us to access **both key and value** together:

for key, value in english_spanish.items():

print("Got", key, "that maps to", value)

Output:

Got three that maps to

tres Got two that maps to

dos Got one that maps to

uno

Using in and not in with Dictionaries

The in operator checks **only keys**, not values.

Examples

"one" in english_spanish

True

"six" in english_spanish

False

"tres" in english_spanish

False (because "tres" is a value, not a key)

Avoiding KeyError

Accessing a missing key causes a runtime

error: english_spanish["dog"]

Error:

KeyError: 'dog'

Safe Practice

Always check before

accessing: if "dog" in

english_spanish:

print(english_spanish["dog"])

- keys(), values(), and items() return **view objects**
- Iterating over a dictionary iterates over its **keys**
- items() is best when you need both key and value
- in checks only for **keys**
- Accessing a non-existent key causes KeyError
- Dictionary order should not be relied upon

3.1.3 Aliasing and Copying

Dictionaries in Python are **mutable**, meaning their contents can be changed after creation. Because of this, we must be careful about **aliasing**—when two variables refer to the **same dictionary object**.

What Is Aliasing?

Aliasing happens when **two variables point to the same object in memory**.

Example

opposites = {"up": "down", "right": "wrong", "yes": "no"}

alias = opposites

Here:

- alias and opposites refer to **the same dictionary**
- No new dictionary is created

Effect of Aliasing

If we modify one variable, the change is visible through the other:

alias["right"] = "left"

print(opposites["right"])

Output:

left

This happens because both names point to the **same dictionary**

object. **Copying a Dictionary**

If we want to modify a dictionary **without affecting the original**, we must create a **copy**.

Using the copy() Method (Shallow Copy)

copy = opposites.copy()

- copy refers to a **new dictionary**
- The key–value pairs are duplicated

- The original dictionary remains unchanged when copy is modified

Modifying the Copy

```
copy["right"] = "privilege"
print(opposites["right"])
```

Output:

left

This confirms:

- copy is independent
- Changes to copy do not affect opposites

Important Note: Shallow Copy

The copy() method creates a **shallow copy**:

- The dictionary itself is new
- But if values inside the dictionary are **mutable objects** (like lists), they are still shared

(Deep copying will be discussed later.)

Operation	Result
alias = opposites	Both refer to same dictionary
Modify alias	Affects opposites
copy = opposites.copy()	Creates new dictionary
Modify copy	Original unchanged

- Dictionaries are mutable
- Aliasing occurs when two variables refer to the same object
- Changes through one alias affect the other
- Use .copy() to preserve the original dictionary
- .copy() creates a shallow copy

3.1.4 Counting Letters

Previously, we counted how many times **one specific letter** appeared in a string. A more powerful and general task is to build a **frequency table**—that is, to count **how many times each letter appears** in a string.

Such frequency tables are useful in many areas, including:

- Text analysis
- Spell checking
- Data compression (common letters can use shorter codes)

Using a Dictionary to Count Letters

Dictionaries are ideal for this problem because they map:

- **keys** → letters
- **values** → number of times each letter appears

Example

```
letter_counts = {}
for letter in "Mississippi":
    letter_counts[letter] = letter_counts.get(letter, 0) + 1
```

Result

```
{'M': 1, 's': 4, 'p': 2, 'i': 4}
```

How This Code Works (Step by Step)

- Start with an **empty dictionary**: letter_counts = {}
- Traverse the string one character at a time
- For each letter:
  - letter_counts.get(letter, 0)
    - returns the current count if the letter exists
    - returns 0 if the letter is not yet in the dictionary
- Add 1 to the count and store it back in the dictionary

  This avoids KeyError and keeps the code clean.

Why get() Is Important

Without get() you would need extra checks like:

```
if letter in letter_counts:
    letter_counts[letter] += 1
else:
    letter_counts[letter] = 1
```

Using get() makes the solution **shorter and clearer**.

Displaying the Frequency Table in Alphabetical Order

Dictionaries themselves are not meant to be ordered. To display results alphabetically, we can:

1. Convert dictionary items into a list
2. Sort the list
3. Print the result

Example

```
letter_items = list(letter_counts.items())
letter_items.sort()
print(letter_items)
```

Output

```
[('M', 1), ('i', 4), ('p', 2), ('s', 4)]
```

Why list() Is Needed

- letter_counts.items() returns a **view object**
- Views cannot be sorted directly
- Converting it to a list allows us to use the sort() method

3.1.5 Glossary

call graph A graph consisting of nodes which represent function frames (or invocations), and directed edges (lines with arrows) showing which frames gave rise to other frames.

dictionary A collection of key:value pairs that maps from keys to values. The keys can be any immutable value, and the associated value can be of any type.

immutable data value A data value which cannot be modified. Assignments to elements or slices (sub parts) of immutable values cause a runtime error.

key A data item that is mapped to a value in a dictionary. Keys are used to look up values in a dictionary. Each key must be unique across the dictionary.

key:value pair One of the pairs of items in a dictionary. Values are looked up in a dictionary by key.

mapping type A mapping type is a data type comprised of a collection of keys and associated values. Python's only

built-in mapping type is the dictionary. Dictionaries implement the associative array abstract data type.

memo Temporary storage of precomputed values to avoid duplicating the same computation.

mutable data value A data value which can be modified. The types of all mutable values are compound types. Lists

and dictionaries are mutable; strings and tuples are not.

3.2 Shape

One of the most important properties of an array is its **shape**.
The **shape** of an array tells us **how many dimensions** it has and **how many elements** exist along each dimension.

In NumPy, the shape is accessed using the **.shape attribute**.

1D Array (One-Dimensional Array)

A **1D array** is like a simple list of values.

Example

```
import numpy as np

a = np.array([2, 3,

8]) print(a.shape)
```

Output

```
(3,)
```

Meaning

- The array has **1 dimension**
- It contains **3 elements**
- The comma (3,) indicates it is a **tuple**, even for 1D arrays

2D Array (Two-Dimensional Array)

A **2D array** looks like a table with **rows and columns**.

Example

```
b = np.array([
   [2, 3, 8],
   [4, 5, 6]
])
print(b.shape)
```

Output

```
(2, 3)
```

Meaning

- **2 rows**
- **3 columns**
- Shape is written as (rows, columns)

Understanding Shape

Values Shape	Meaning
(3,)	1D array with 3 elements
(2, 3)	2D array with 2 rows and 3 columns (height, width) Grayscale image
(height, width, 3)	Color (RGB) image

3D Arrays and Images

- **Grayscale image** → 2D array (pixels)
- **Color image (RGB)** → 3D array

Each pixel in a color image contains **3 values**:

- Red
- Green
- Blue

So a color image has

shape: (height, width, 3)

Example:

(1080, 1920, 3)

This means:

- 1080 rows (height)
- 1920 columns (width)
- 3 color channels (RGB)

Why Shape Is Important

Knowing the shape is essential for:

- Image processing
- Machine learning models
- Matrix operations
- Debugging errors
- Understanding data layout

Many NumPy operations **depend on matching shapes**.

3.3 Slicing

In NumPy, **slicing** is used to select specific elements from an array.
For **1D arrays**, slicing works **exactly like Python lists**.
For **2D and higher-dimensional arrays**, slicing allows us to select **rows, columns, or sub-arrays**.

Slicing a 1D NumPy Array

```
import numpy as np


a = np.array([2, 3, 8])
```

Indexing

```
a[2]
```

Output

```
8
```

Slicing

```
a[1:]
```

Output

```
array([3, 8])
```

Same behavior as Python lists.

Slicing a 2D NumPy Array

```
b =
   np.array([
   [2, 3, 8],
   [4, 5, 6]
])
```

This array has shape (2, 3) → **2 rows, 3 columns**

Selecting a Row

```
b[1]
```

Output

```
array([4, 5, 6])
```

Explanation:

- b[1] selects the **1st row** (remember: counting starts at 0)
- The result is still a **1D NumPy array**

Selecting an Individual Element

b[1][2]

Output

6

This can be written more cleanly as:

b[1, 2]

Same result
Preferred NumPy style

Selecting a Column (Very Important)

To select a column, we use the **colon :**

operator. b[:, 1]

Output

array([3, 5])

How to Read This

- : → select **all rows**
- 1 → select **column index 1**

So:

b[:, 1] → all rows, column 1

Understanding the Colon :

In NumPy slicing:

Expression	Meaning
:	all elements along that dimension
b[row, col]	element at (row, col)
b[row, :]	entire row
b[:, col]	entire column

Why NumPy Slicing Is Powerful

NumPy slicing allows:

- Easy row/column selection
- Fast data access
- Clean mathematical operations
- Efficient image and matrix

processing This is heavily used in:

- Machine Learning
- Data Science
- Image Processing
- Scientific Computing

Important Reminder About Indexing

- Python uses **zero-based indexing**
- So:
    - 0th row → first row
    - 1th row → second row
      (The term *1th* is used intentionally to remind you of the **0th element**.)

3.4 Masking

Masking is one of the most powerful features of NumPy.
It allows us to **select and modify elements of an array based on a condition**, **without using loops**. In simple terms:

A **mask** is a Boolean array (True / False) created by applying a condition to a NumPy array.

Creating a Mask

import numpy as np

a = np.array([230, 10, 284, 39, 76])

cutoff =

200 a >

cutoff

Output

array([True, False, True, False, False])

What is happening here?

- NumPy compares **each element** with 200
- The result is a **Boolean array**
- True → condition satisfied
- False → condition not

satisfied This Boolean array is called

a **mask**.

Using a Mask to Modify Values

Now we set all values **greater than 200 to**

zero: a[a > cutoff] = 0

print(a)

Output

array([0, 10, 0, 39, 76])

Key Line (Most Important)

a[a > cutoff] = 0

How to read this

- a > cutoff → creates a Boolean mask
- a[mask] → selects only elements where the mask is True
- = 0 → assigns 0 to all those positions

Only the elements **230** and **284** were replaced
Other values remain unchanged

Why Masking Is Powerful

Without masking, we would need a

loop: new_a = []

```
for x in a:
    if x > cutoff:
        new_a.append(0)
    else:
        new_a.append(
```

x) a =

np.array(new_a)

Problems with looping

- More code
- Slower execution
- Difficult for large data
- Extremely messy for images (2D or 3D

arrays) Masking solves all of this in **one line**.

Masking with Images (Why It Matters)

In image processing:

- Images are **2D or 3D NumPy arrays**
- Masking allows:
    - Thresholding
    - Noise removal
    - Background removal
    - Feature extraction

Example:

image[image < 50] = 0

This removes dark pixels instantly — no loops required.

Common Masking Patterns

Operation	**Example**
Set high values to zero	a[a > 200] = 0
Select values	a[a < 50]
Replace values	a[a == 10] = 99
Combine conditions	a[(a > 10) & (a < 100)]

Masking in NumPy is a technique where Boolean conditions are used to select and modify array elements efficiently without explicit loops.

3.5 Broadcasting

Broadcasting is a powerful NumPy feature that allows **element-wise operations on arrays of**

different shapes, **without explicitly copying data**.

In simple terms:

NumPy automatically expands (stretches) smaller arrays so that their shapes become compatible for computation.

Basic Example of Broadcasting

import numpy as

np a = np.array([

[0, 1],

[2, 3],

[4, 5]

])

b = np.array([10,

100]) a * b

Output

array([[0, 100],

[20, 300],

[40, 500]])

Why does this work?

- a has shape **(3, 2)** → 3 rows, 2 columns
- b has shape **(2,)** → 1D array with 2 elements

NumPy **automatically stretches b** to behave

like: [[10, 100],

[10, 100],

[10, 100]]

Then multiplication happens **element-wise**.

Broadcasting Rule 1: Size-1 Dimensions Can Stretch

- NumPy can **stretch only dimensions of size 1**
- If an array has fewer dimensions, NumPy **adds size-1 dimensions implicitly**

In the example:

- b (2,) → treated as (1, 2)
- (1, 2) can stretch to (3, 2) to match a

Broadcasting Rule 2: Compare Dimensions from Right to Left

When operating on two arrays:

1. Compare shapes **from the last dimension**
2. Dimensions must:
    - Be equal, **or**
    - One of them must be 1
3. Otherwise → **broadcasting fails**

Example Where Broadcasting Fails

```
c =
np.array([
[0, 1, 2],
[3, 4, 5]
])
b = np.array([10, 100])
c * b
```

Error

ValueError: operands could not be broadcast together with shapes (2,3) (2,)

Why does this fail?

- c → shape (2, 3)
- b → shape (2,) → treated as (1,

2) Comparing last dimensions:

- 3 vs 2
- Neither dimension is 1, so **no stretching possible**

Fixing Broadcasting with None (or np.newaxis)

We can **explicitly add a**

dimension: c * b[:, None]

What happens here?

- b[:, None] changes shape from (2,) → (2, 1)
- Now shapes are:

- c → (2, 3)
- b[:, None] → (2, 1)

The size-1 dimension can stretch:

Output

array([[0, 10, 20],
[300, 400, 500]])

Understanding None in

Broadcasting Expression	**Resulting Shape**
b	(2,)
b[None, :]	(1, 2)
b[:, None]	(2, 1)

This is extremely useful for:

- Matrix math
- Normalization
- Image processing
- Machine learning tensors

Why Broadcasting Is Important

Broadcasting:

- Avoids loops
- Saves memory
- Makes NumPy fast
- Enables clean mathematical code

Used heavily in:

- Data Science
- Machine Learning
- Image Processing
- Scientific Computing

Broadcasting in NumPy is a mechanism that allows element-wise operations on arrays of different shapes by automatically expanding dimensions of size one.

3.6 dtype

In NumPy, **dtype** stands for **data type**.
It defines **how values are stored in memory**, including:

- whether the values are **integers or floats**
- whether they are **signed or unsigned**
- how many **bits** are used to store each

value Examples:

- int8, int16, int32, int64
- uint8, uint16
- float32, float64

Understanding Bits and Ranges

A **bit** can be either 0 or 1.

Unsigned 8-bit integer (uint8)

- Uses **8 bits**
- Total values = $2^8 = 256$
- Range: **0 to 255**
- No negative numbers

Signed 8-bit integer (int8)

- Also uses **8 bits**
- Range: **−128 to +127**
- Can store negative values

Larger Data

Types int64

- Uses **64 bits**
- Range:
- −9,223,372,036,854,775,808
- to
- +9,223,372,036,854,775,807
- Default integer type on most **64-bit systems**

Why Bigger Is NOT Always Better

You might think:

"Why not always use the biggest integer

type?" That is usually **not a good idea**,

because:

- Bigger dtype → **more memory usage**
- Large arrays (images, ML data) can consume **huge memory**
- Slower performance in some cases

Memory example

If your values never exceed 100:

- uint8 → efficient
- int64 → wastes memory

Default dtype is fine

Optimize only when memory becomes a problem

Overflow: A Critical Concept

Example with uint8

```
import numpy as np

a = np.array([200], dtype='uint8')

a + a
```

Output

```
array([144], dtype=uint8)
```

Why not 400?

- Maximum uint8 value = 255
- 200 + 200 = 400
- 400 does **not fit** in uint8
- NumPy **wraps around** (modulo 256)

400 − 256 = 144

This behavior is called **overflow**.

Fixing Overflow

Use a larger data type:

a = np.array([200],

dtype='uint16') a + a

Output
array([400], dtype=uint16)
Result fits
No overflow

dtype and Images (Very Important)

Most images (.jpg, .png) use:

dtype = uint8

Each pixel is stored as:

(R, G, B) → (0–255, 0–255, 0–255)

Examples:

- (0, 0, 0) → Black
- (255, 0, 0) → Red

Why Images Can Break Without Care

If you add an image to

itself: image + image

You might expect:

"The image becomes brighter"

But instead:

- Pixel values overflow
- Colors wrap around
- You get **noise and distortion**

Correct approach

Convert before operations:

image = image.astype(np.uint16)
result = image + image

dtype in NumPy specifies the data type, size, and range of values stored in an array and directly affects memory usage and numerical behavior.

3.7 About Files

While a program is running, all its data is stored in **RAM (Random Access Memory)**.
RAM is **fast**, but it is **volatile**, which means:

- When the program ends
- Or when the computer is shut down

All data in RAM is lost

Why Files Are Needed

To **save data permanently**, information must be stored on **non-volatile storage**, such as:

- Hard disk
- USB drive
- SSD
- CD/DVD

Data stored on these devices is kept in **files**.

What Is a File?

A **file** is:

- A **named location** on non-volatile storage
- Used to store data **permanently**

By using files, programs can:

- Save data for later use
- Load data when the program runs again
- Share data between different programs

Files and Programs

Programs can:

- **Read** data from files
- **Write** data to files

This allows data to **persist between program runs**.

Notebook Analogy (Very Important for Understanding)

Working with files is like working with a **notebook**:

Notebook	File
Must be opened	File must be
opened Can read or write	Can read or
write	
Has a current position	File pointer keeps track of
position Must be closed	File must be closed

You can:

- Read the notebook from start to end
- Skip pages
- Write new

information The same

applies to files.

Opening a File

To open a file, we must specify:

- **File name**
- **Mode of operation**

Common modes:

- 'r' → read
- 'w' → write
- 'a' → append

Example:

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

Closing a File

When we finish working with a file, it must be **closed**:

```
file.close()
```

Closing a file:

- Frees system resources
- Ensures data is properly saved

Files are named locations on non-volatile storage that allow programs to store and retrieve data permanently.

3.8 Writing Our First File

To store data permanently, a Python program can **write data to a file**. This section explains how to **create a file and write text into it**.

Basic Example: Writing to a File

```
with open("test.txt", "w") as myfile:
    myfile.write("My first file written from
    Python\n") myfile.write("----------\n")
    myfile.write("Hello, world!\n")
```

Understanding Each Line

Line 1: Opening the File

```
with open("test.txt", "w") as myfile:
```

- open() creates a **file handle**
- "test.txt" → name of the file
- "w" → write mode
- myfile → variable that stores the file handle

What does write mode ("w") do?

- If test.txt **does not exist** → it is created
- If test.txt **already exists** → it is erased and replaced

Writing Data to the File

```
myfile.write("Hello, world!\n")
```

- write() sends text to the file
- \n creates a **new line**
- Data is written **exactly as provided**

In real programs, writing is often done inside **loops** to store many lines.

Automatic File Closing with with

- The with statement **automatically closes the file**
- File is closed:
    - When execution reaches the end of the block
    - Even if an error occurs (except power failure)

This makes with **safer and cleaner** than manually calling close().

What Is a File Handle?

A **file handle** is a reference that lets the program interact with a file.

Important Concept

- The **file handle is NOT the file**
- It is a **control object** used to access the file

File Handle Analogy (Very Important for Exams)

A file handle is like a **TV remote control**:

TV Remote	**File Handle**
You press buttons	Program calls methods
Remote controls TV	Handle controls file
Action happens on TV	Action happens on disk

We often say:

- "Close the file" instead of
- "Close the file handle"

This is called **abstraction**.

Writing a file in Python involves opening a file in write mode, using a file handle to write data, and closing the file to save the changes permanently.

3.9 Reading a File Line-at-a-Time

Once a file exists on disk, a Python program can **open it for reading** and process its contents **one line at a time**.
This approach is memory-efficient and widely used for large files.

Basic Example: Reading a File Line by Line

```
with open("test.txt", "r") as my_new_handle:
    for the_line in my_new_handle:
```

```
print(the_line, end="")
```

Understanding Each Part

Opening the File (Read Mode)

with open("test.txt", "r") as my_new_handle:

- "r" → read mode
- The file must already exist
- my_new_handle is the **file handle**
- with ensures the file is **closed automatically**

Reading Line-by-Line Using a for Loop

for the_line in my_new_handle:

- The file handle is **iterable**
- Each iteration reads **one full line**
- Each line includes the **newline character (\n)**

Printing Without Extra Newlines

print(the_line, end="")

- print() normally adds a newline
- Each line already ends with \n
- end="" prevents **double newlines**

Why This Pattern Is Important

This pattern is commonly used because:

- It is **simple and readable**
- It handles **large files efficiently**
- Each line can be processed independently

In real programs, instead of just printing, you might:

- Split the line into fields
- Parse data
- Call other functions (e.g., send emails, process logs)

What Happens If the File Does Not Exist?

open("wharrah.txt", "r")

Error

FileNotFoundError: [Errno 2] No such file or directory

Explanation

- Read mode ("r") **requires the file to exist**
- Python raises an error if it cannot find the file

Reading a file line-at-a-time in Python is done by opening the file in read mode and iterating over the file handle, where each iteration returns one line.

3.10 Turning a File into a List of Lines

Sometimes it is more convenient to **read an entire file into memory at once** and work with its contents as a **list of lines**.

This is especially useful when we want to:

- Sort the lines
- Search or filter lines
- Modify data and write it back to a file

Example Use Case

Suppose we have a file friends.txt where:

- Each line contains a friend's name and email address
- We want to **sort the lines alphabetically**
- Then write the sorted result to a new file

Reading All Lines into a List

```
with open("friends.txt", "r") as
   input_file: all_lines =
   input_file.readlines()
```

What readlines() Does

- Reads **all lines** from the file
- Returns a **list of strings**
- Each string represents **one line**, including the newline character

\n Example:

["Ali ali@email.com\n", "Zara zara@email.com\n", "Iqra iqra@email.com\n"]

Sorting the Lines

all_lines.sort()

- Sorts the list **alphabetically**
- Sorting happens **in memory**
- No file access is needed during sorting

Writing the Sorted Lines to a New File

```
with open("sortedfriends.txt", "w") as output_file:
    for line in all_lines:
        output_file.write(line)
```

What Happens Here

- A new file sortedfriends.txt is created
- Each line from the sorted list is written back to the file
- The original newline characters ensure correct formatting

Why Use readlines() Instead of a Loop?

We *could* read the file line-by-line and build the list manually, but:

- readlines() is **shorter**
- **Cleaner code**
- Less error-prone
- Provided directly by Python for convenience

Important Notes

- readlines() loads the **entire file into memory**
- Best suited for **small or medium-sized files**
- For very large files, reading line-by-line is more memory-efficient
- with ensures files are properly closed

Turning a file into a list of lines involves reading all lines using readlines(), processing them as a list, and optionally writing the results back to a file.

3.11 Reading the Whole File at Once

Another way to work with files in Python is to **read the entire file contents into a single string**. This method is useful when:

- We are **not interested in line-by-line structure**
- We want to perform **string processing**, such as:
    - Counting words
    - Searching text
    - Removing punctuation
    - Text analysis

Basic Example: Reading the Entire File

with open("somefile.txt") as f:

```
content = f.read()

words = content.split()

print("There are {0} words in the
file.".format(len(words)))
```

Understanding Each

Step Opening the File

with open("somefile.txt") as f:

- No mode is specified
- Default mode is **read mode ("r")**
- File must exist
- with ensures automatic file closing

Reading the Entire File

content = f.read()

- Reads **all text** from the file
- Stores it as **one large string**
- Includes spaces, newlines, and tabs

Splitting into Words

words = content.split()

- split() breaks the string into a **list of words**
- Splits on **any whitespace** (spaces, tabs, newlines)

Counting Words

len(words)

- Counts how many words are present
- Result is printed using format()

When to Use This Method

Use read() when:

- File size is **small or moderate**
- You want to use **string methods**
- Line boundaries are **not important**

Avoid read() for:

- Very large files (memory-heavy)

File Paths (Important Concept)

The example assumes:

- somefile.txt is in the **same directory** as the Python program

  If not, you must specify the path.

Examples

Windows

"C:\\temp\\somefile.txt"

Linux / macOS

"/home/jimmy/somefile.txt"

You can also use **relative paths**:
"data/somefile.txt"

Reading the whole file at once involves loading the complete file contents into a string using `read()` and then processing it using string operations.

3.12 An Example – File Filters

Many real-world programs **read a file line-by-line**, apply **simple rules**, and **write selected or modified lines** to another file.
Such programs are called **filters** because they *filter out* unwanted content.

Typical filter tasks

- Remove comment lines
- Number lines
- Insert blank lines after a fixed count
- Extract specific columns
- Keep only lines containing a substring

Example: Filtering Out Comment Lines

The following function copies one file to another **but omits lines that start with #** (commonly used for comments):

```
def filter(oldfile, newfile):
    with open(oldfile, "r") as infile, open(newfile, "w") as
        outfile: for line in infile:
            # Put any processing logic here
            if not line.startswith('#'):
                outfile.write(line)
```

How This Program Works (Line by Line)

Line 1: Function Definition

```
def filter(oldfile, newfile):
```

- oldfile → source file (input)
- newfile → destination file (output)

Line 2: Opening Two Files at Once

```
with open(oldfile, "r") as infile, open(newfile, "w") as
outfile:
```

- Opens **input file** in read mode
- Opens **output file** in write mode
- Both files are safely closed when the with block ends

Line 3: Read Input File Line-by-Line

for line in infile:

- Reads one line at a time
- Efficient for large files
- Each line includes its newline character

Lines 4–6: Filtering Logic

if not line.startswith('#'):

outfile.write(line)

- startswith('#') checks if a line is a comment
- not reverses the condition
- Only **non-comment lines** are written to the output file

Why This Is Called a Filter

- It **passes through** some lines
- It **blocks** others based on a rule
- The structure is reusable for many tasks by changing the condition

Advantages of This Pattern

- Simple and readable
- Memory-efficient
- Works well for large text files
- Easy to extend with additional logic

Common Variations You Can

Build # Keep only lines containing

a word if "error" in line:

Remove blank

lines if line.strip()

!= "":

Number lines

outfile.write(f"{count}:

{line}")

A file filter is a program that reads a file line-by-line, applies selection or transformation rules, and writes the result to another file.

3.14 Directories

Files on non-volatile storage (hard disks, SSDs, USB drives) are organized using a **file system**. A file system consists of:

- **Files** → store data
- **Directories (folders)** → containers that hold files and other

directories This creates a **hierarchical structure**, similar to a tree.

Current Directory (Working Directory)

- When a program **creates a file**, it is placed in the **current directory**
- When a program **opens a file for reading**, Python looks for it in the **current directory** by default

If the file is **not** in the current directory, we must provide its **path**.

What Is a Path?

A **path** specifies the exact location of a file by listing the directories leading to it.

Unix/Linux Example

```
wordsfile = open("/usr/share/dict/words", "r")
wordlist = wordsfile.readlines()
print(wordlist[:6])
```

Explanation of the path:

- / → top-level (root) directory
- usr → subdirectory inside /
- share → subdirectory inside usr
- dict → subdirectory inside share
- words → file inside dict

This example reads all lines from the file and prints the first few entries.

Windows Paths

Windows paths can be written as:

"c:/temp/words.txt"

or

"c:\\temp\\words.txt"

Why double backslashes?

- In Python strings, \ is an **escape character**
- To represent a literal backslash, we must write \\

So these two strings have the **same length and meaning**.

Reserved Characters

- / and \ are **directory separators**
- They **cannot be used** as part of a file name
- They only indicate directory boundaries

Why Use os.path? (Very Important)

Different operating systems use different path separators:

- Unix/Linux/macOS → /
- Windows → \

To avoid problems, Python provides the **os.path module**.

Recommended Way

```
import os
path = os.path.join("directory", "filename")
```

- On Unix/Linux → "directory/filename"
- On Windows → "directory\\filename"

Portable

Safer
No manual slash handling

Why This Matters

Using os.path:

- Makes code **cross-platform**
- Prevents string-escaping mistakes
- Simplifies file and directory

handling The os.path module also provides tools to:

- Check if a file exists

- Get file names and extensions
- Work with absolute and relative paths

Directories organize files in a hierarchical file system, and file paths specify the exact location of a file within that structure.

3.14 Fetching Something from the Web

Python programs can **download data from the Internet** just like a web browser.
This allows programs to:

- Fetch documents (text, images, files)
- Read online data directly
- Process web content automatically

Python offers multiple libraries to do this. Two common approaches are:

1. **urllib.request** (standard library)
2. **requests** (external, more powerful and user-friendly)

Using urllib.request to Download a File

Example: Download a Web File and Save It Locally

```
import urllib.request
url = "http://xml.resource.org/public/rfc/txt/rfc793.txt"
destination_filename = "rfc793.txt"


urllib.request.urlretrieve(url, destination_filename)
```

What This Does

- Downloads the content from the URL
- Saves it directly to a local file
- The file is created in the **current directory**

Why This Is Powerful

- A **single function call** downloads any web resource
- Works for text files, images, PDFs, etc.

Important Conditions for urlretrieve

Before this works correctly:

- The **URL must exist** (check in a browser)

- You must have **write permission** in the directory
- If behind a **proxy with authentication**, extra configuration may be needed
- Always **verify downloaded content** before using it

Web content can change, disappear, or even become malicious.

Using the requests Module (Recommended)

The requests module is:

- Easier to use
- More flexible
- Not included by default (must be installed) **Example: Read Web Content into a String** import requests

```
url = "http://xml.resource.org/public/rfc/txt/rfc793.txt"
response = requests.get(url)
print(response.text)
```

Explanation

- requests.get(url) sends an HTTP request
- The server returns a **response object**
- response.text contains the entire page as a **string**

Reading Web Content Line-by-Line

```
import requests

url =
"http://xml.resource.org/public/rfc/txt/rfc793.txt"
response = requests.get(url)
for line in response:
    print(line)
```

- Allows incremental processing
- Useful for large files
- Similar to reading a local file line-by-line

Why Verification Is Critical

When fetching data from the web:

- Websites can change without warning
- Content can be replaced or removed
- Malicious content is possible

Best practice:
Always check that downloaded data is what your program expects **before processing or displaying it**.

Fetching data from the web in Python involves sending an HTTP request to a URL and processing the server's response using libraries such as urllib or requests.

3.15 Glossary

delimiter A sequence of one or more characters used to specify the boundary between separate parts of text.

directory A named collection of files, also called a folder. Directories can contain files and other directories, which are referred to as subdirectories of the directory that contains them.

file A named entity, usually stored on a hard drive, floppy disk, or CD-ROM, that contains a stream of characters.

file system A method for naming, accessing, and organizing files and the data they contain.

handle An object in our program that is connected to an underlying resource (e.g. a file). The file handle lets our program manipulate/read/write/close the actual file that is on our disk.

mode A distinct method of operation within a computer program. Files in Python can be opened in one of four modes:

read ("r"), write ("w"), append ("a"), and read and write ("+").

non-volatile memory Memory that can maintain its state without power. Hard drives, flash drives, and rewritable compact disks (CD-RW) are each examples of non-volatile memory.

path A sequence of directory names that specifies the exact location of a file.

text file A file that contains printable characters organized into lines separated by newline characters.

volatile memory Memory which requires an electrical current to maintain state. The main memory or RAM of a computer is volatile. Information stored in RAM is lost when the computer is turned off.