Computer Programming I Lecture Notes

Piya Limcharoen

Version October 15, 2025at 8:19am

Contents

Copyright Notice

This lecture note was authored by Piya Limcharoen. Artificial Intelligence (AI) was utilized as a tool for language enhancement and grammatical correction; however, all intellectual property, original concepts, and the structure of this work belong exclusively to the author.

Terms of Use

This document is provided for educational purposes only and is intended for the exclusive use of students currently enrolled in this course.

The reproduction, distribution, public display, or transmission of this material, in whole or in part, in any form or by any means—electronic, mechanical, photocopying, recording, or otherwise—is strictly prohibited without the prior written permission of the author. Students may not share, upload, or distribute these notes outside of the official class environment.

Warning

Any unauthorized use of this material will be considered academic misconduct and will be subject to university disciplinary procedures. Furthermore, such an act may constitute an infringement of copyright law and is subject to legal action.

Lecture 1:

File Handling for Program Persistence

Lecture 1: File Handling for Program Persistence

1.1 Transient vs. Persistent Programs

Most of the programs we have encountered so far are **transient**[1]. A transient program executes for a limited duration and typically produces some output during its execution, but once it terminates, all data stored in memory is lost. If the program is run again, it starts from an initial (empty) state as if it had never been executed before.

Example: A simple Python script that takes two numbers as input, adds them, prints the result, and then exits is a transient program. When the script finishes, the input values and the computed result are not stored anywhere for later use.

In contrast, **persistent**[2] programs are designed to preserve data beyond the duration of their execution. Such programs typically store information in permanent storage devices (like hard drives or solid-state drives) and can restore their previous state when restarted.

Example: An operating system is a persistent program—it runs continuously while the computer is on, managing resources and maintaining logs. Similarly, a web server (such as Apache or Nginx) is persistent, running indefinitely and storing access logs and configuration data to ensure continuity even after restarts.

[1] *Transient* means temporary or short-lived; something that lasts only for a brief period of time.

[2] *Persistent* means continuing to exist or remain active for a prolonged time; in computing, it refers to data or programs that maintain their state between executions.

Persistence in programs is often achieved through **file handling**, where data is written to and read from files stored on disk. Files serve as a simple yet powerful mechanism for data storage.

1.2 Text and Binary Files

Files are generally categorized into two types: **text files** and **binary files**.

- **Text files** store data in human-readable form, using standard character encodings such as ASCII or UTF-8. Examples include `.txt`, `.csv`, and `.py` files. Text files are convenient for simple data storage but may be inefficient for complex data structures.
- **Binary files**, on the other hand, store data in a format readable only by programs. They can represent any kind of information—images, videos, compiled code, or serialized objects—without converting it to text. Examples include `.jpg`, `.exe`, and `.dat` files. Binary storage is more compact and faster for reading/writing large or structured data.

In Python, data persistence can be achieved in several ways, depending on the format of the stored data.

- For **text-based persistence**, programs can store data in human-readable files such as plain text (`.txt`), comma-separated values (`.csv`), or JavaScript Object Notation (`.json`) files. These formats are easy to inspect and edit manually. For instance, Python's built-in `open()` function can be used to read and write text files, and modules like `csv` and `json` simplify working with structured data.

- For **binary persistence**, Python provides the `pickle` module, which allows programs to serialize (convert) complex Python objects—such as lists, dictionaries, or custom classes—into a binary format that can be written to a file. The same module can later deserialize (restore) the objects from the file back into memory. This mechanism enables programs to save their internal state and resume from that state later, effectively bridging the gap between transient and persistent behavior.

1.3 What is a File?

In computing, a **file** is a named collection of related data stored on a secondary storage device such as a hard drive, SSD, or USB drive. Files serve as the primary means of storing and retrieving data outside a program's runtime memory, allowing information to persist even after the program or computer is shut down.

Files can contain any form of data—text, numbers, images, audio, video, or program instructions—and are organized and managed by the operating system through a **file system**. Each file is identified by a unique name and often an extension (e.g., `.txt`, `.csv`, `.exe`) that indicates its type or intended use.

The process of interacting with files is known as **File I/O (Input/Output)**:

- **Input (I)** refers to reading data from an external source into a program. For example, loading configuration data from a text file or importing saved game progress.
- **Output (O)** refers to writing data from a program to an external destination, such as saving a report, writing logs, or exporting user preferences.

File I/O enables communication between a program's transient memory (RAM) and persistent storage (disk), making it essential for data retention and long-term storage.

A typical file-handling operation follows three main steps:

1. **Opening** a file — establishes a connection between the program and the file. In Python, this is done using the `open()` function, specifying both the file name and the mode (e.g., read, write, or append).
2. **Reading from or Writing to** the file — depending on the mode, the program either retrieves existing data (input) or sends new data to the file (output). Python provides methods such as `read()`, `readline()`, and `write()` for these operations.
3. **Closing** the file — releases system resources and ensures that all written data is properly saved. In Python, this is typically done using the `close()` method or automatically managed with a `with` statement for safer handling.

Understanding file operations and I/O is fundamental to developing persistent programs, as it allows data to exist beyond a single program's execution and supports long-term information management.

1.4 Working with Filenames and Paths

Correctly managing file paths is crucial for writing robust programs that work across different operating systems (Windows, macOS, Linux). Improper handling of paths can lead to errors such as "file not found" or "invalid path." Python's `pathlib` module provides an object-oriented and platform-independent way to work with file paths safely and efficiently.

1.4.1 Absolute vs. Relative Paths

- **Absolute Path**: A full path that specifies a file's location from the root of the file system. It uniquely identifies the file's position in the directory structure.
  - Windows: `C:\Users\Ku\Documents\report.txt`
  - macOS/Linux: `/home/ku/documents/report.txt`
- **Relative Path**: A path that specifies a file's location relative to the current working directory (CWD). It is shorter and more portable but depends on where the program is executed.
  - Example: `data/sales.csv` or `../images/logo.png`

1.4.2 Path Handling in Python

1.4.2.1 Getting the Current Working and Home Directories

```python
from pathlib import Path

# Get the Current Working Directory (CWD)
cwd = Path.cwd()
print(f"Current Working Directory: {cwd}")

# Get the user's home directory
home_dir = Path.home()
print(f"Home Directory: {home_dir}")
```

Sample Output (on macOS):

```
Current Working Directory: /Users/youruser/projects/my_app
Home Directory: /Users/youruser
```

1.4.2.2 Creating Platform-Independent Paths

The **pathlib** module lets you build paths using the **/** operator, which automatically applies the correct separator for the host operating system ("\" on Windows, "/" on Unix-like systems).

```python
from pathlib import Path

# Build a path to a file in a subdirectory of the CWD
data_file_path = Path.cwd() / "data" / "users.csv"
print(f"Constructed Path: {data_file_path}")

# You can also join Path objects with other Path objects or strings
reports_dir = Path("reports")
monthly_report = reports_dir / "january" / "sales.txt"
print(f"Report Path: {monthly_report}")
```

Note: The `/` operator only works with `Path` objects. Attempting to use it with normal strings causes a `TypeError`, as shown below.

```
>>> 'spam' / 'bacon'
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unsupported operand type(s) for /: 'str' and 'str'
>>>
```

Explanation:

Strings in Python do not support the division operator (`/`). The `Path` class overloads this operator to simplify path concatenation in a way that is operating-system–independent.

Using plain strings to create paths manually (e.g., `"C:\folder\file.txt"`) is error-prone and non-portable, because separators differ between operating systems. Always use `pathlib.Path()` to ensure compatibility and safety.

1.4.2.3 Deconstructing Path Components

Once you have a `Path` object, you can easily extract its individual parts. This is useful for getting a filename, its extension, or its parent folder without manual string manipulation.

```python
from pathlib import Path

p = Path.cwd() / "data" / "users.csv"

print(f"Original Path: {p}")
print(f"Parent Directory: {p.parent}")
print(f"File Name: {p.name}")
print(f"File Stem (name without extension): {p.stem}")
print(f"File Suffix (extension): {p.suffix}")
```

Sample Output:

```
Original Path: /Users/youruser/projects/my_app/data/users.csv
Parent Directory: /Users/youruser/projects/my_app/data
File Name: users.csv
File Stem (name without extension): users
File Suffix (extension): .csv
```

1.4.2.4 Checking Path Validity

Before reading or writing a file, it's good practice to check whether the path exists and whether it refers to a file or directory.

```python
from pathlib import Path

path = Path("data/users.csv")

# Check if the path exists
print(f"Does '{path}' exist? {path.exists()}")

# Check if it's a file
print(f"Is '{path}' a file? {path.is_file()}")

# Check if it's a directory
print(f"Is '{path}' a directory? {path.is_dir()}")
```

1.4.2.5 Creating Directories

Often, you need to ensure a directory exists before you can write a file into it. `Path.mkdir()` is the modern way to create directories.

```
from pathlib import Path

# Define a path for a new directory to store output files
output_dir = Path.cwd() / "output"
print(f"Attempting to create directory: {output_dir}")

# Create the directory
# parents=True: Creates parent directories if they don't exist.
# exist_ok=True: Does not raise an error if the directory already exists.
output_dir.mkdir(parents=True, exist_ok=True)

print(f"Does '{output_dir}' now exist? {output_dir.exists()}")
```

Explanation: Using `mkdir(parents=True, exist_ok=True)` is a robust pattern that safely creates a directory structure without crashing if the script is run more than once.

1.4.2.6 Listing Directory Contents

While **`pathlib`** is ideal for constructing and managing paths, changing the current working directory (CWD) is performed using the **`os`** module. After changing the directory, you can use **`Path.iterdir()`** to list all files and subdirectories in the new location.

```python
import os
from pathlib import Path

print(f"Original CWD: {Path.cwd()}")

# Define the target directory path
target_dir = Path.home() / "Documents"

# Change the current working directory if it exists
if target_dir.is_dir():
    os.chdir(target_dir)
    print(f"New CWD: {Path.cwd()}")

    # List contents of the new current directory
    print("\nContents of the new directory:")
    for item in Path.cwd().iterdir():
        if item.is_dir():
            print(f"[DIR]  {item.name}")
        else:
            print(f"       {item.name}")
```

```
21 else:
22     print(f"Directory '{target_dir}' does not exist.")
```

1.5 Text Files vs. Binary Files

Files on a computer are stored as sequences of bytes. How a program interprets these bytes depends on the file type.

- **Text Files (.txt, .py, .csv, .html)**: The bytes in the file represent characters, following an encoding scheme like UTF-8 or ASCII. When you open a text file in Python (e.g., with mode `'r'`), Python automatically decodes the bytes into strings for you. These files are human-readable.
- **Binary Files (.jpg, .mp3, .exe, .pkl)**: The bytes represent data that is not plain text, such as image pixels, audio samples, or serialized Python objects. To work with these files, you must open them in a binary mode (e.g., `'rb'` or `'wb'`), which gives you direct access to the raw bytes without any decoding.

1.5.1 Expanding on File Modes

When you open a file with `open()`, the **mode** string specifies your intent—reading, writing, or both—and whether you are handling text or binary data. Using the correct mode is crucial for preventing errors and data corruption.

The mode can be a combination of one character from the first group and one from the second:

1. **Operation Type**: `'r'` (read), `'w'` (write), `'a'` (append), `'x'` (exclusive creation)
2. **Data Type**: `'t'` (text, default), `'b'` (binary)
3. **Modifier**: `'+'` (update - read and write)

Key Points:

- **Truncation:** Be careful with `'w'` and `'w+'` modes, as they will **erase all existing content** in a file upon opening.
- **Binary Mode (`b`):** This is essential for non-text files. Use it for images (`.jpg`), executables (`.exe`), and serialized data from `pickle` (`.pkl`). Attempting to open these files in text mode will result in errors or corrupted data.
- **Exclusive Creation (`x`):** This mode is a safe way to create a new file without accidentally overwriting an existing one.

Here is a comprehensive table of common file modes:

Mode	Description	Behavior If File Exists	Behavior If File Doesn't Exist
`'r'`	Read Text (Default)	Reads from the start.	Raises `FileNotFoundError`.
`'w'`	Write Text	**Truncates** (empties) the file.	**Creates** a new file.
`'a'`	Append Text	Appends to the end of the file.	**Creates** a new file.
`'x'`	Exclusive Creation	Raises `FileExistsError`.	**Creates** a new file.
`'rb'`	Read Binary	Reads from the start.	Raises `FileNotFoundError`.
`'wb'`	Write Binary	**Truncates** the file.	**Creates** a new file.
`'r+'`	Read and Write	Pointer is at the start.	Raises `FileNotFoundError`.
`'w+'`	Write and Read	**Truncates** the file first.	**Creates** a new file.

1.6 Reading from Text Files

1.6.1 Reading Files Using `open()` and `close()`

In Python, files can be opened and closed manually using the built-in **`open()`** and **`close()`** functions. Python's file handling model, using **`open()`** and **`close()`**, is conceptually similar to the **`fopen()`** and **`fclose()`** functions found in the C programming language. However, Python offers a more modern and safer approach with the with statement.

```
# Open a file for reading (equivalent to fopen in C)
f = open('poem.txt', 'r', encoding='utf-8')

# Read the entire content
content = f.read()
print(content)

# Close the file to free system resources (like fclose in C)
f.close()
```

Output:

```
Roses are red,
Violets are blue.
```

Explanation:

- `open()` returns a `file object`, which acts as a handle to the file.
- The second argument specifies the mode:
  - `'r'` – read (file must exist)
  - `'w'` – write (creates or overwrites a file)
  - `'a'` – append (writes at the end of the file)
- `f.close()` releases system resources and ensures that all data is saved.

Important: If you forget to call `close()`, the file remains open until Python's garbage collector closes it automatically. This may cause memory leaks, file locks, or incomplete writes.

Note on encoding: If you omit the `encoding` parameter (e.g., `open('poem.txt', 'r')`), Python uses the system default encoding, which varies by platform:

- On Windows, the default is often `'cp1252'`.
- On macOS and Linux, the default is usually `'utf-8'`.

Using the default may cause errors when reading files created on a different system or containing special characters. Therefore, it is best practice to explicitly specify `encoding='utf-8'` or another appropriate encoding.

1.6.2 Using `with open()` — Context Manager Style

A safer and more modern approach is to use a **`with`** statement, which automatically closes the file after the block ends, even if an error occurs.

```
1 with open('poem.txt', 'r', encoding='utf-8') as f:
2     content = f.read()
3     print(content)
4 # File is automatically closed when exiting the 'with' block
```

Advantages:

- Eliminates the need to call **`close()`** manually.
- Prevents data loss and resource leaks if an exception occurs.
- Makes the code shorter, cleaner, and more Pythonic.

1.6.3 Reading Entire File or Line by Line

Assume a file `poem.txt` contains:

```
Roses are red,
Violets are blue.
```

```
# Reading the entire file into one string
with open('poem.txt', 'r', encoding='utf-8') as f:
    content = f.read()
    # content is 'Roses are red,\nViolets are blue.'

# Reading the file line by line into a list
with open('poem.txt', 'r', encoding='utf-8') as f:
    lines = f.readlines()
    # lines is ['Roses are red,\n', 'Violets are blue.']
```

1.6.4 Parsing Lines with `split()`

Each line in a text file may contain multiple pieces of data. Use `.strip()` to remove whitespace (like `n`) and `.split()` to separate the line into a list of strings.

```
data_line = "item:price:quantity\n"
parts = data_line.strip().split(':')
# parts is now ['item', 'price', 'quantity']
```

1.6.5 Reading and Nested Splitting

Some files contain multiple records per line, requiring nested splitting. For example, `inventory.txt` contains:

```
apple,1.20,10;banana,0.80,5;cherry,2.50,12
```

Each product is separated by a semicolon (`;`) and each field by a comma (`,`).

```
with open('inventory.txt', 'r', encoding='utf-8') as f:
    line = f.readline().strip()
    # Split line into product entries
    products = line.split(';')
    for product in products:
        name, price, qty = product.split(',')
        print(f"Name: {name}, Price: {price}, Quantity: {qty}")
```

Sample Output:

```
Name: apple, Price: 1.20, Quantity: 10
Name: banana, Price: 0.80, Quantity: 5
Name: cherry, Price: 2.50, Quantity: 12
```

Explanation: This is called **nested splitting**: first split by records (`;`), then by fields (`,`). It is commonly used for structured text data.

1.6.6 Reading Structured Text Files (CSV/TSV)

CSV (Comma-Separated Values) and TSV (Tab-Separated Values) are widely used formats for tabular data. While you can parse them manually with `split()`, Python's built-in `csv` module provides a safer and more robust approach, handling quotes, delimiters, and special characters automatically.

Example CSV File: `students.csv`

```
Name,Major,GPA,GraduationYear
Alice,Computer Science,3.9,2024
Bob,Physics,3.5,2023
Charlie,Mathematics,3.8,2025
```

```python
import csv

# Reading CSV file
with open('students.csv', 'r', newline='', encoding='utf-8') as f:
    reader = csv.reader(f)

    # Read header
    header = next(reader)
    print(f"Header: {header}")

    # Read each row
    for row in reader:
        print(f"Name: {row[0]}, Major: {row[1]}, GPA: {row[2]}, Graduation Year: {row
          [3]}")
```

Sample Output:

```
Header: ['Name', 'Major', 'GPA', 'GraduationYear']
Name: Alice, Major: Computer Science, GPA: 3.9, Graduation Year: 2024
Name: Bob, Major: Physics, GPA: 3.5, Graduation Year: 2023
Name: Charlie, Major: Mathematics, GPA: 3.8, Graduation Year: 2025
```

Example TSV File: employees.tsv

```
ID  Name    Department  Salary
101 John    Accounting  55000
102 Mary    HR  60000
103 Peter   IT  70000
```

```python
import csv

# Reading TSV file (tab-delimited)
with open('employees.tsv', 'r', newline='', encoding='utf-8') as f:
    reader = csv.reader(f, delimiter='\t')

    # Read header
    header = next(reader)
    print(f"Header: {header}")

    # Read each row
    for row in reader:
        print(f"ID: {row[0]}, Name: {row[1]}, Department: {row[2]}, Salary: {row[3]}")
```

Sample Output:

```
Header: ['ID', 'Name', 'Department', 'Salary']
ID: 101, Name: John, Department: Accounting, Salary: 55000
ID: 102, Name: Mary, Department: HR, Salary: 60000
ID: 103, Name: Peter, Department: IT, Salary: 70000
```

Notes:

- For TSV files, set `delimiter='\t'` when creating the reader object.
- Always specify `encoding='utf-8'` to ensure cross-platform compatibility and avoid `UnicodeDecodeError`.
- The `csv` module automatically handles quoted fields and embedded delimiters, which makes it safer than manually splitting lines.

1.7 Writing to Text Files

To write to a text file in Python, you can use mode `'w'` to create or overwrite a file, or `'a'` to append to it. Files can be handled either manually with `open()`/`close()` or using the `with` statement for automatic closure.

1.7.1 File Object Attributes

When a file is opened, Python returns a `file object` with useful attributes:

- `name` – the name of the file.
- `mode` – the mode in which the file was opened (`'r'`, `'w'`, `'a'`, etc.).
- `closed` – Boolean, indicates whether the file is closed.

```
1 f = open('report.txt', 'w')
2 print(f"File Name: {f.name}")
3 print(f"File Mode: {f.mode}")
4 print(f"Is Closed? {f.closed}")
5
6 f.write("Hello, world!\n")
7 f.write("Hello, KU!\n")
8 f.close()
9 print(f"Is Closed after close()? {f.closed}")
```

Output:

```
File Name: report.txt
File Mode: w
Is Closed? False
Is Closed after close()? True
```

Contents of `report.txt`

```
Hello, world!
Hello, KU!
```

1.7.2 Manual Writing with `open()` and `close()`

You can write to a file manually, similar to C-style **`fopen`**/**`fclose`**:

```
# Open a file for writing
f = open('report.txt', 'w', encoding='utf-8')
f.write("This is the first line.\n")
f.write("This line is written manually.\n")
f.close()
```

Contents of `report.txt`

```
This is the first line.
This line is written manually.
```

Note: Specifying **`encoding='utf-8'`** ensures correct handling of Unicode characters.

1.7.3 Writing from a List with `writelines()`

The **`writelines()`** method writes all strings from a list to a file. You must include newline characters yourself.

```
lines_to_write = ["First item\n", "Second item\n", "Third item\n"]
with open('items.txt', 'w', encoding='utf-8') as f:
    f.writelines(lines_to_write)
```

Contents of `items.txt`

```
First item
Second item
Third item
```

Note: Both **`write()`** and **`writelines()`** do not automatically add newline characters, so each string must include a newline character if you want a line break in the output.

1.7.4 Writing Nested Data Structures Manually

Before using CSV or TSV modules, you can write nested lists or dictionaries manually by iterating over them:

```
nested_dict = {
    "Alice": {"Major": "CS", "GPA": 3.9},
    "Bob": {"Major": "Physics", "GPA": 3.5}
}

with open('students_manual.txt', 'w', encoding='utf-8') as f:
    for name, info in nested_dict.items():
        line = f"{name},{info['Major']},{info['GPA']}\n"
        f.write(line)
```

Contents of `students_manual.txt`:

```
Alice,CS,3.9
Bob,Physics,3.5
```

1.7.5 Writing Structured Text Files (CSV/TSV) with `writerow()`

You can write rows to a CSV file using **`csv.writer`** and its **`writerow()`** method. Each row is provided as a list of values corresponding to the columns.

```
import csv

with open('names.csv', 'w', newline='', encoding='utf-8') as csvfile:
    writer = csv.writer(csvfile)

    # Write the header row
    writer.writerow(['first_name', 'last_name'])

    # Write data rows
    writer.writerow(['Baked', 'Beans'])
    writer.writerow(['Lovely', 'Spam'])
    writer.writerow(['Wonderful', 'Spam'])
```

Contents of `names.csv`:

```
first_name,last_name
Baked,Beans
Lovely,Spam
Wonderful,Spam
```

Explanation:

- `writerow()` writes a single row as a list of values.
- The first row can be used as a header, followed by data rows.
- This approach is simpler when your data is already organized as lists rather than dictionaries.

1.7.6 Writing Structured Text Files (CSV/TSV) with `writerows()`

For structured data, Python's **`csv`** module simplifies writing CSV or TSV files.

```python
import csv

# Data to write (list of lists)
student_data = [
    ["Name", "Major", "GPA"],
    ["Charlie", "Literature", 3.7],
    ["David", "Engineering", 3.8]
]

# Writing to a CSV file
with open('new_students.csv', 'w', newline='', encoding='utf-8') as f:
    writer = csv.writer(f)
    writer.writerows(student_data)

# Writing to a TSV file (tab-delimited)
with open('new_students.tsv', 'w', newline='', encoding='utf-8') as f:
    writer = csv.writer(f, delimiter='\t')
    writer.writerows(student_data)

print("CSV and TSV files have been created.")
```

Contents of `new_students.csv`:

```
Name,Major,GPA
Charlie,Literature,3.7
David,Engineering,3.8
```

Contents of `new_students.tsv`:

```
Name    Major   GPA
Charlie Literature  3.7
David   Engineering 3.8
```

Explanation:

- `writerows()` writes multiple rows at once; each row is a list of values.
- `delimiter='\t'` specifies tab separation for TSV files.
- Using `newline=''` avoids adding extra blank lines on Windows.
- Always specify `encoding='utf-8'` for cross-platform consistency.

1.8 Working with Binary Files: Serialization with `pickle`

1.8.1 What is Serialization?

Serialization is the process of converting a Python object (like a list or dictionary) into a byte stream that can be stored in a file. This allows you to save complex data structures and load them back later, preserving their state. Python's standard library for this is **`pickle`**.

1.8.2 Saving Objects with `pickle.dump()`

To save an object, open the file in binary write mode (**`'wb'`**).

```python
import pickle

# A complex Python object
user_prefs = {"font_size": 12, "theme": "dark", "bookmarks": [101, 205, 350]}

with open('preferences.pkl', 'wb') as f:
    pickle.dump(user_prefs, f)

print("Preferences object saved to preferences.pkl")
```

Security Warning: The **`pickle`** module is not secure. Never unpickle data received from an untrusted source, as it can be crafted to execute malicious code.

1.8.3 Loading Objects with `pickle.load()`

To load the object back, open the file in binary read mode (`'rb'`).

```python
import pickle

with open('preferences.pkl', 'rb') as f:
    loaded_prefs = pickle.load(f)

print("--- Loaded Preferences ---")
print(loaded_prefs)
print(f"Theme: {loaded_prefs['theme']}")
```

Output:

```
--- Loaded Preferences ---
{'font_size': 12, 'theme': 'dark', 'bookmarks': [101, 205, 350]}
Theme: dark
```

Lecture 2:

Exception and Exception Handling

Lecture 2: Exception and Exception Handling

2.1 Why Do We Need Exception Handling?

Before diving into the technical details, let's think about why exception handling is so crucial. Imagine your program is a highly organized workshop.

- **Normal Flow:** Your skilled workers (the code) are assembling products smoothly.
- **An Exception:** Suddenly, a machine breaks down (a runtime error occurs). Without a plan, the entire assembly line halts, work is left unfinished, and chaos ensues. Your whole workshop (the program) shuts down.
- **Exception Handling ('try...except'):** This is your workshop's emergency plan. When a machine breaks, an alarm sounds (`try` detects an error) and a specialized repair crew (`except`) immediately handles the broken machine. The main production might pause, but the workshop itself doesn't shut down. The repair crew can fix the problem or log it for later, allowing other parts of the workshop to continue running.
- **Cleanup ('finally'):** This is the safety protocol that runs no matter what—like turning off the main power to the broken machine's area. This cleanup happens whether the machine was fixed or not, ensuring the workshop is left in a safe state.

Exception handling transforms your program from a fragile process that breaks at the first sign of trouble into a robust system that can anticipate, manage, and recover from unexpected problems.

2.2 What is an Exception?

In programming, an **exception** is an error event that occurs during the execution of a program that disrupts its normal flow. Unlike syntax errors, which are detected by the interpreter before the program is run, exceptions occur at runtime. When the Python interpreter encounters an error it cannot handle, it "raises" an exception.

If the exception is not handled by the program, the program will terminate and display a traceback message, which provides information about where the error occurred.

Common built-in exceptions in Python include:

- `TypeError`: Raised when an operation or function is applied to an object of an inappropriate type.
- `ValueError`: Raised when an operation or function receives an argument that has the right type but an inappropriate value.
- `ZeroDivisionError`: Raised when the second argument of a division or modulo operation is zero.
- `FileNotFoundError`: Raised when a file or directory is requested but doesn't exist.
- `IndexError`: Raised when a sequence subscript is out of range.
- `KeyError`: Raised when a dictionary key is not found.

2.2.1 Examples of Unhandled Exceptions

Example 1: `ZeroDivisionError`

```
# This code will cause a ZeroDivisionError
numerator = 10
denominator = 0
result = numerator / denominator
print(result)
```

Output (Traceback):

```
Traceback (most recent call last):
  File "<stdin>", line 3, in <module>
ZeroDivisionError: division by zero
```

Example 2: ValueError

```
# This code will cause a ValueError
age_str = "twenty"
age_int = int(age_str) # Cannot convert 'twenty' to an integer
print(age_int)
```

Output (Traceback):

```
Traceback (most recent call last):
  File "<stdin>", line 2, in <module>
ValueError: invalid literal for int() with base 10: 'twenty'
```

Example 3: IndexError

```
# This code will cause an IndexError
my_list = [10, 20, 30]
print(my_list[3]) # Index 3 is out of bounds for a list of size 3
```

Output (Traceback):

```
Traceback (most recent call last):
  File "<stdin>", line 2, in <module>
IndexError: list index out of range
```

2.3 The `try...except` Block

To prevent a program from crashing due to an exception, you can use **exception handling**. The core mechanism for this in Python is the `try...except` block.

The basic idea is to place the code that might raise an exception inside the `try` block. If an exception occurs, the rest of the `try` block is skipped, and the code inside the corresponding `except` block is executed. If no exception occurs, the `except` block is ignored.

2.3.1 Examples of `try...except`

Example 1: Handling a `ZeroDivisionError`

```
numerator = 10
denominator = 0

try:
    result = numerator / denominator
    print("This will not be printed.")
except ZeroDivisionError:
    print("Error: Cannot divide by zero.")

print("The program continues to run.")
```

Output:

```
Error: Cannot divide by zero.
The program continues to run.
```

Example 2: Handling a `ValueError`

```python
try:
    user_input = input("Enter a number: ")
    number = int(user_input)
    print(f"You entered the number {number}.")
except ValueError:
    print("Invalid input. Please enter a valid integer.")
```

Sample Output (with invalid input):

```
Enter a number: abc
Invalid input. Please enter a valid integer.
```

Example 3: Handling a `FileNotFoundError`

```
try:
    with open("non_existent_file.txt", "r") as f:
        content = f.read()
        print(content)
except FileNotFoundError:
    print("Error: The file could not be found.")
```

Output:

```
Error: The file could not be found.
```

2.4 Handling Specific Exceptions

It is good practice to handle specific exceptions rather than using a generic `except` block. This allows your program to respond appropriately to different types of errors. You can have multiple `except` blocks to handle various exceptions, or handle multiple exceptions in a single block.

2.4.1 Examples of Handling Specific Exceptions

Example 1: Multiple `except` Blocks

```
1  def process_data(data, index):
2      try:
3          value = int(data[index])
4          result = 100 / value
5          print(f"The result is: {result}")
6      except IndexError:
7          print("Error: Index is out of bounds.")
8      except ValueError:
9          print("Error: The data at the given index is not a valid integer.")
10     except ZeroDivisionError:
11         print("Error: The value at the given index is zero, cannot divide.")
12
13 my_list = ["10", "20", "0", "abc"]
14 process_data(my_list, 2)  # ZeroDivisionError
15 process_data(my_list, 3)  # ValueError
16 process_data(my_list, 5)  # IndexError
```

Output:

```
Error: The value at the given index is zero, cannot divide.
Error: The data at the given index is not a valid integer.
Error: Index is out of bounds.
```

Example 2: Grouping Multiple Exceptions

```python
def calculate_average(numbers):
    try:
        total = sum(numbers)
        count = len(numbers)
        average = total / count
        print(f"The average is {average}.")
    except (TypeError, ZeroDivisionError) as e:
        print(f"Could not calculate average. Error: {e}")

calculate_average([10, 20, 'thirty']) # Raises TypeError
calculate_average([])                 # Raises ZeroDivisionError
```

Output:

```
Could not calculate average. Error: unsupported operand type(s) for +: 'int' and 'str'
Could not calculate average. Error: division by zero
```

Example 3: Catching the Base `Exception` Class

```python
import sys
try:
    # A risky operation
    f = open('myfile.txt')
    s = f.readline()
    i = int(s.strip())
    result = 10 / i
except Exception as e:
    # This will catch any exception that inherits from Exception
    print(f"An unexpected error occurred: {e}")
    print(f"Error type: {type(e).__name__}")
```

Sample Output (if myfile.txt contains '0'):

```
An unexpected error occurred: division by zero
Error type: ZeroDivisionError
```

2.5 The `else` and `finally` Clauses

The `try...except` block can be extended with optional `else` and `finally` clauses.

- **`else` Clause**: The code inside the `else` block is executed **only if no exceptions are raised** in the `try` block.
- **`finally` Clause**: The code inside the `finally` block is **always executed**, regardless of whether an exception occurred or not. It is ideal for cleanup actions.

2.5.1 Examples of `else` and `finally`

Example 1: Basic Usage

```python
def divide_numbers(a, b):
    try:
        print("Attempting division...")
        result = a / b
    except ZeroDivisionError:
        print("Caught a ZeroDivisionError!")
    else:
        print(f"Division successful! Result is {result}")
    finally:
        print("This 'finally' block is always executed.")

divide_numbers(10, 2); print("-" * 20)  # No exception
divide_numbers(10, 0) # An exception occurs
```

Output:

```
Attempting division...
Division successful! Result is 5.0
This 'finally' block is always executed.
--------------------
Attempting division...
Caught a ZeroDivisionError!
This 'finally' block is always executed.
```

Example 2: File Handling with `finally`

```
f = None # Initialize f to ensure it exists in the finally block
try:
    f = open("important_data.txt", "w")
    f.write("This is a critical message.")
    # Simulate an error after writing
    # int("abc") # Uncomment this line to see finally run after an error
except Exception as e:
    print(f"An error occurred: {e}")
else:
    print("File written successfully.")
finally:
    if f: # Check if the file was opened
        f.close()
        print("File has been closed.")
```

Output:

```
File written successfully.
File has been closed.
```

Example 3: Return Value from `finally` A `return` statement in a `finally` block will override any `return` in the `try` or `except` blocks.

```
1  def check_status():
2      try:
3          print("Trying to return 'Success'")
4          return "Success from try"
5      except:
6          return "Failure from except"
7      finally:
8          print("Finally block is executed, returning 'Complete'")
9          return "Complete from finally"
10
11 print(check_status())
```

Output:

```
1 Trying to return 'Success'
2 Finally block is executed, returning 'Complete'
3 Complete from finally
```

2.6 Raising Exceptions

You can manually raise an exception using the **`raise`** keyword. This is useful for signaling an error condition based on your program's logic.

2.6.1 Examples of Raising Exceptions

Example 1: Raising a `ValueError`

```
1 def set_age(age):
2     if age < 0 or age > 120:
3         raise ValueError("Age must be between 0 and 120.")
4     print(f"Age set to {age}")
5 
6 try:
7     set_age(150)
8 except ValueError as e:
9     print(f"Error: {e}")
```

Output:

```
1 Error: Age must be between 0 and 120.
```

Example 2: Raising a TypeError

```
1  def add_to_list(items, item):
2      if not isinstance(items, list):
3          raise TypeError("First argument must be a list.")
4      items.append(item)
5      return items
6
7  my_tuple = (1, 2)
8  try:
9      add_to_list(my_tuple, 3)
10 except TypeError as e:
11     print(f"Error: {e}")
```

Output:

```
1  Error: First argument must be a list.
```

Example 3: Re-raising an Exception

Sometimes you may want to catch an exception, perform an action (like logging), and then re-raise it to let a higher-level handler deal with it.

```
def process_file(filename):
    try:
        with open(filename, 'r') as f:
            print(f.read())
    except FileNotFoundError as e:
        print(f"Logging error: Could not find {filename}.")
        raise # Re-raise the caught exception

try:
    process_file("data.txt")
except FileNotFoundError:
    print("Error handler: The program cannot continue without the file.")
```

Output:

```
Logging error: Could not find data.txt.
Error handler: The program cannot continue without the file.
```

2.7 The `assert` Statement

The **`assert`** statement is a debugging aid that tests a condition. If the condition is true, it does nothing, and the program continues. If the condition is false, it raises an **`AssertionError`** with an optional message.

Assertions are meant for internal self-checks and should not be used for handling expected runtime errors (like invalid user input), as assertions can be globally disabled.

Syntax:

```
1 assert condition, "Optional error message if condition is false"
```

2.7.1 Examples of using `assert`

Example 1: Checking a Value

```
1 # This assertion will pass silently
2 x = 10
3 assert x > 0, "x should be positive"
4 print("Assertion passed.")
5
6 # This assertion will fail and raise an AssertionError
7 y = -5
8 assert y > 0, "y should be positive"
9 print("This line will not be executed.")
```

Output:

```
Assertion passed.
Traceback (most recent call last):
  File "<stdin>", line 7, in <module>
AssertionError: y should be positive
```

Example 2: Sanity Check in a Function

```python
def calculate_discount(price, discount_percent):
    assert 0 <= discount_percent <= 100, "Discount must be between 0 and 100"
    return price * (1 - discount_percent / 100)

# This works fine
print(calculate_discount(50, 20))

# This will raise an AssertionError
print(calculate_discount(50, 110))
```

Output:

```
40.0
Traceback (most recent call last):
  File "<stdin>", line 8, in <module>
  File "<stdin>", line 2, in calculate_discount
AssertionError: Discount must be between 0 and 100
```

Example 3: Checking a Type

```python
def process_list(items):
    assert isinstance(items, list), "Input must be a list"
    for item in items:
        print(item)

# This works fine
process_list([1, 2, 3])

# This will raise an AssertionError
process_list((4, 5, 6)) # A tuple, not a list
```

Output:

```
1
2
3
Traceback (most recent call last):
  File "<stdin>", line 9, in <module>
  File "<stdin>", line 2, in process_list
AssertionError: Input must be a list
```

2.8 `assert` vs. `raise`: When to Use Each

It's common for beginners to confuse when to use an **`assert`** statement versus raising an exception with **`raise`**. The distinction is critical for writing correct and maintainable code. Here is a direct comparison:

Feature	**`assert` Statement**	**`raise` Statement**
Purpose	**Debugging.** Used to check for internal conditions that should *never* happen. Catches programmer errors.	**Error Handling.** Used to signal an expected, possible error condition that can occur at runtime (e.g., bad user input, file not found).
Audience	For the developer. It says, "I believe this condition to be true, and if it's not, my program has a bug."	For the user or caller of the code. It says, "You have provided invalid data or an unrecoverable situation has occurred."
When it's Active	Can be globally **disabled** in production by running Python with the **`-O`** (optimize) flag. They should not be relied upon for application logic.	Is **always active** and is a core part of the program's control flow and logic.
Exception Type	Always raises an **`AssertionError`**.	Can raise any type of exception, including built-in ones (**`ValueError`**, **`TypeError`**) or custom exceptions.

Golden Rule: Use **`assert`** to check for bugs in your own code. Use **`raise`** to signal errors to the person using your code.

2.9 Creating Custom Exceptions

For application-specific problems, you can create custom exceptions by defining a new class that inherits from the base `Exception` class. This makes your code more readable and your error handling more specific.

2.9.1 Examples of Custom Exceptions

Example 1: `InsufficientFundsError`

```
class InsufficientFundsError(Exception):
    """Raised when a withdrawal is attempted with insufficient funds."""
    def __init__(self, balance, amount):
        message = f"Attempted to withdraw {amount} with a balance of only {balance}"
        super().__init__(message)

def withdraw(balance, amount):
    if amount > balance:
        raise InsufficientFundsError(balance, amount)
    return balance - amount

try:
    withdraw(balance=100, amount=500)
except InsufficientFundsError as e:
    print(f"Transaction failed: {e}")
```

Output:

```
Transaction failed: Attempted to withdraw 500 with a balance of only 100
```

Example 2: InvalidEmailError

```python
class InvalidEmailError(ValueError):
    """Raised when an email format is invalid."""
    pass # Inherits behavior from ValueError

def send_email(email, message):
    if "@" not in email:
        raise InvalidEmailError(f"'{email}' is not a valid email address.")
    print(f"Email sent to {email}.")

try:
    send_email("not-an-email", "Hello!")
except InvalidEmailError as e:
    print(f"Error: {e}")
```

Output:

```
Error: 'not-an-email' is not a valid email address.
```

Example 3: PasswordTooShortError

```python
class PasswordTooShortError(Exception):
    def __init__(self, length, min_length=8):
        message = f"Password is too short. Required: {min_length}, Provided: {length}"
        super().__init__(message)

def set_password(password):
    if len(password) < 8:
        raise PasswordTooShortError(len(password))
    print("Password has been set successfully.")

try:
    set_password("12345")
except PasswordTooShortError as e:
    print(f"Could not set password. Reason: {e}")
```

Output:

```
Could not set password. Reason: Password is too short. Required: 8, Provided: 5
```

2.10 Summary: Best Practices for Exception Handling

To write clean, robust, and professional Python code, follow these key principles for exception handling:

- **Be Specific, Not General.** Always catch the most specific exception you can. Avoid bare `except:` blocks, as they can hide real bugs.

```python
# Good: Specific and clear
try:
    value = int("a")
except ValueError:
    print("Invalid number format.")

# Bad: Hides all errors, even ones you didn't expect
try:
    # ... code ...
except: # This could hide a TypeError, NameError, etc.
    print("Something went wrong.")
```

- **Keep `try` Blocks Small.** Only wrap the specific lines of code that you expect might raise an exception inside the `try` block. This makes your code easier to read and debug.
- **Use `finally` for Cleanup.** For actions that *must* be completed—like closing a file or a network connection—place the code in a `finally` block to guarantee its execution.
- **Don't Use Exceptions for Normal Control Flow.** Exceptions are for exceptional, unexpected events. If you can easily check for a condition with an `if` statement, that is more efficient and clearer than using a `try...except` block.
- **Create Custom Exceptions for Your Application.** When you need to signal an error that is specific to your program's domain (e.g., `InsufficientFundsError`), create a custom exception class. This makes your program's error conditions explicit and self-documenting.