

# Advanced Programming

## Chapter 3: Input-Output and Exception Handling

# What is a file?

- A file is a collection of data that is stored on secondary storage like a disk or a thumb drive
- accessing a file means establishing a connection between the file and the program and moving data between the two

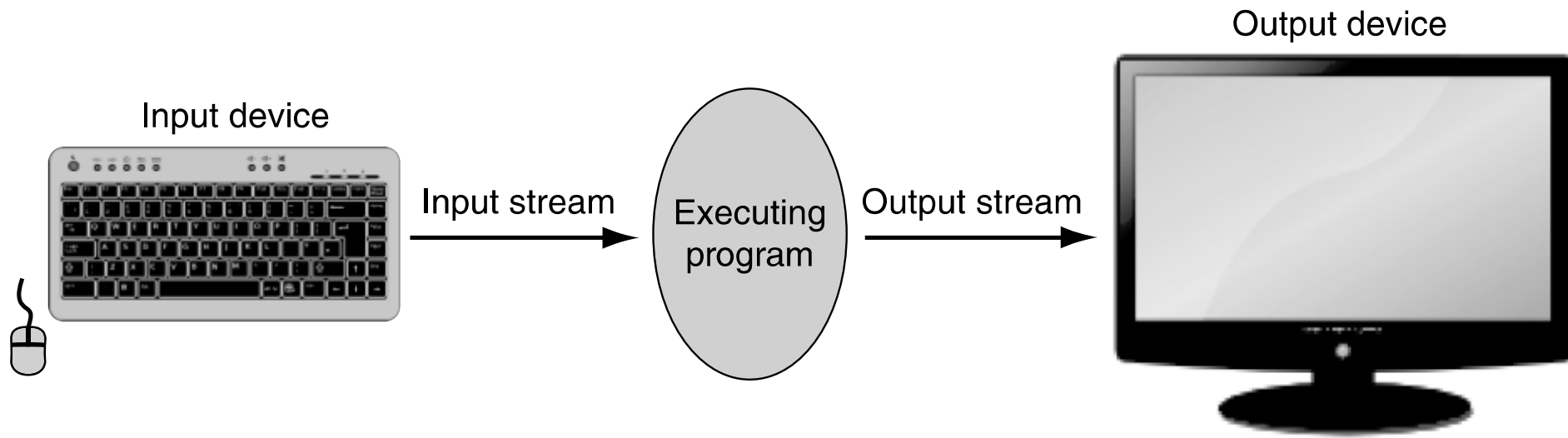
# Two types of files

Files come in two general types:

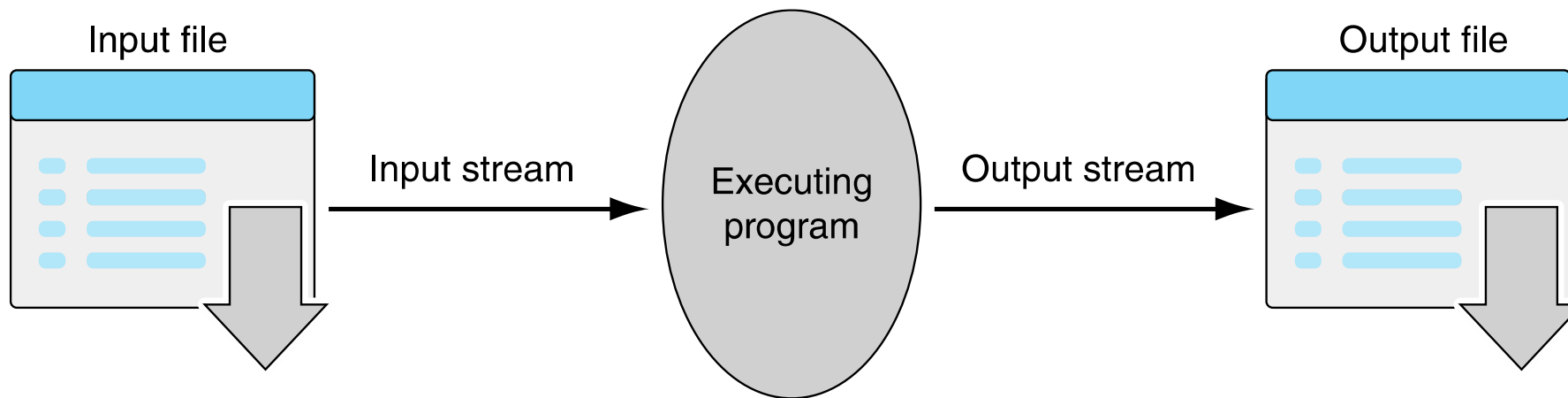
- *text files*. Files where control characters such as `"/n"` are translated. These are generally human readable
- *binary files*. All the information is taken directly without translation. Not readable and contains non-readable info.

# File Objects or stream

- When opening a file, you create a file object or file stream that is a connection between the file information on disk and the program.
- The stream contains a buffer of the information from the file, and provides the information to the program



a) Standard input and output



b) File input and output

**FIGURE 5.1** Input-output streams.

# File Handling with Exception Handling

File input/output operations, error handling, and exception management.

## Example Code 1: process\_file(file\_path)

Case 1: File exists with content

Case 2: File does not exist

Case 3: File is empty

Case 4: Unexpected error

```
def process_file(file_path):  
    try:  
        with open(file_path, 'r') as file:  
            print(f"Opening file: {file_path}")  
            content = file.read()  
            if not content:  
                raise ValueError("File is empty")  
            lines = content.splitlines()  
            print("File content processed successfully.")  
            return lines  
    except FileNotFoundError:  
        print(f"Error: File {file_path} not found.")  
    except ValueError as ve:  
        print(f"Error: {ve}")  
    except Exception as e:  
        print(f"An unexpected error occurred: {e}")  
    finally:  
        print("Finished processing the file.")
```

# Buffering

- Reading from a disk is very slow. Thus the computer will read a lot of data from a file in the hopes that, if you need the data in the future, it will be buffered in the file object.
- This means that the file object contains a copy of information from the file called a cache (pronounced "cash")

# Making a file object

```
my_file = open("my_file.txt", "r")
```

- `my_file` is the file object. It contains the buffer of information. The open function creates the connection between the disk file and the file object. The first quoted string is the file name on disk, the second is the mode to open it (here, "`r`" means to read)



# Where is the disk file?

- When opened, the name of the file can come in one of two forms:
- `"file.txt"` assumes the file name is file.txt and it is located in the current program directory
- `"c:\bill\file.txt"` is the fully qualified file name and includes the directory information

# Different modes

Mode	How Opened	File Exists	File Does Not Exist
'r'	read-only	Opens that file	Error
'w'	write-only	Clears the file contents	Creates and opens a new file
'a'	write-only	File contents left intact and new data appended at file's end	Creates and opens a new file
'r+'	read and write	Reads and overwrites from the file's beginning	Error
'w+'	read and write	Clears the file contents	Creates and opens a new file
'a+'	read and write	File contents left intact and read and write at file's end	Creates and opens a new file

**TABLE 5.1** File Modes

# Careful with write modes

- Be careful if you open a file with the `'w'` mode. It sets an existing file's contents to be empty, destroying any existing data.
- The `'a'` mode is nicer, allowing you to write to the end of an existing file without changing the existing contents

# Text files use strings

- If you are interacting with text files (which is all we will do in this book), remember that *everything is a string*
  - everything read is a string
  - if you write to a file, you can only write a string

# Getting File Contents

- Once you have a file object:
- `fileObject.read()` - reads the entire contents of the file as a string and returns it. It can take an optional argument integer to limit the read to N bytes, that is `fileObject.read(N)`
- `fileObject.readline()` - delivers the next line as a string

# More File Reads

- `fileObject.readlines()` - returns a single list of all the lines from the file
- `for line in fileObject:` - iterator to go through the lines of a file

# writing to a file

Once you have created a file object, opened for writing, you can use the print command

- you add `file=file` to the print command

```
# open file for writing:  
#     creates file if it does not exist  
#     overwrites file if it exists  
>>> temp_file = open("temp.txt", "w")  
>>> print("first line", file=temp_file)  
>>> print("second line", file=temp_file)  
>>> temp_file.close()
```

```
input_file = open("input.txt", "r")
output_file = open("output.txt", "w")

for line_str in input_file:
    new_str = ''
    line_str = line_str.strip()           # get rid of carriage return
    for char in line_str:
        new_str = char + new_str        # concat at the left (reverse)
    print(new_str, file=output_file)    # print to output_file

    # include a print to shell so we can observe progress
    print('Line: {:12s} reversed is: {:s}'.format(line_str, new_str))
input_file.close()
output_file.close()
```



# Writing to a File with Exception Handling

File input/output operations, error handling, and exception management.

## Example Code 2:

`write_to_file(file_path, data)`

Case 1: Successful write

Case 2: Permission denied

Case 3: Invalid data (non-string in list)

Case 4: Unexpected error

```
def write_to_file(file_path, data):
    try:
        with open(file_path, 'w') as file:
            for line in data:
                if not isinstance(line, str):
                    raise TypeError("All data must be in string format.")
                file.write(line + '\n')
            print(f"Data successfully written to {file_path}")
    except PermissionError:
        print(f"Error: Permission denied to write to {file_path}.")
    except TypeError as te:
        print(f"Error: {te}")
    except Exception as e:
        print(f"An unexpected error occurred while writing to file: {e}")
    finally:
        print("File writing process completed.")
```

# close

When the program is finished with a file, we `close` the file

- flush the buffer contents from the computer to the file
- tear down the connection to the file
- `close` is a method of a file obj  
`file_obj.close()`
- All files should be closed!

# Scenario 1: Merging Multiple Files and Handling Different File Formats

You have multiple files of varying formats (some are text files, and some are CSV files). Your task is to read these files, merge their contents, and save the combined output into a new file.

## Requirements:

- 1.If the file format is .txt, read it as plain text.
- 2.If the file format is .csv, read it as CSV and convert it to a list of strings.
- 3.Ignore any empty files.
- 4.Handle situations where one or more files do not exist.
- 5.Ensure that file permissions are handled gracefully.
- 6.Raise a custom exception if all the files are empty.

```
import os
import csv

class AllFilesEmptyError(Exception):
    """Custom exception raised when all files are empty."""
    pass

def read_file(file_path):
    try:
        if file_path.endswith(".txt"):
            with open(file_path, 'r') as file:
                content = file.read().strip()
                if content:
                    return content.splitlines()
        elif file_path.endswith(".csv"):
            with open(file_path, 'r') as file:
                reader = csv.reader(file)
                content = list(reader)
                if content:
                    return [' '.join(row) for row in content]
        else:
            print(f"Skipping unsupported file format: {file_path}")
            return []
    except FileNotFoundError:
        print(f"File not found: {file_path}")
        return []
    except PermissionError:
        print(f"Permission denied for file: {file_path}")
        return []
```

```
def merge_files(file_list, output_file):
    all_lines = []
    for file in file_list:
        lines = read_file(file)
        all_lines.extend(lines)

    if not all_lines:
        raise AllFilesEmptyError("All files are empty!")

    try:
        with open(output_file, 'w') as out_file:
            for line in all_lines:
                out_file.write(line + '\n')
            print(f"Merged content written to {output_file}")
    except Exception as e:
        print(f"An error occurred while writing to {output_file}: {e}")

# Test with example files
file_list = ['file1.txt', 'file2.csv', 'file3.txt']
merge_files(file_list, 'merged_output.txt')
```

## Scenario 2: Log File Analysis and Reporting

You have a large log file (logfile.log) where each line contains log data in the format:

[Timestamp] - [Log Level] - [Message]

Your task is to:

- 1.Extract and categorize log messages by their log levels (INFO, WARNING, ERROR).
- 2.Save each category into its own file (info.log, warning.log, error.log).
- 3.Raise a custom exception if no errors are found in the log file.
- 4.Ensure large files are processed efficiently by reading them in chunks.

```
class NoErrorsFoundError(Exception):
    """Custom exception raised when no errors are found in the log file."""
    pass

def process_log_file(log_file_path):
    info_logs = []
    warning_logs = []
    error_logs = []

    try:
        with open(log_file_path, 'r') as log_file:
            while True:
                lines = log_file.readlines(1000) # Read the log file in chunks
                if not lines:
                    break

                for line in lines:
                    if "INFO" in line:
                        info_logs.append(line)
                    elif "WARNING" in line:
                        warning_logs.append(line)
                    elif "ERROR" in line:
                        error_logs.append(line)
```

```
    if not error_logs:
        raise NoErrorsFoundError("No errors found in the log file!")

    save_logs("info.log", info_logs)
    save_logs("warning.log", warning_logs)
    save_logs("error.log", error_logs)
    print("Logs have been categorized and saved.")

except FileNotFoundError:
    print(f"Log file {log_file_path} not found.")
except PermissionError:
    print(f"Permission denied for log file: {log_file_path}.")
except Exception as e:
    print(f"An unexpected error occurred: {e}")

def save_logs(output_file, logs):
    try:
        with open(output_file, 'w') as file:
            file.writelines(logs)
    except Exception as e:
        print(f"Failed to write to {output_file}: {e}")

# Example usage
process_log_file('logfile.log')
```

# Exceptions

## First Cut

# How to deal with problems

- Most modern languages provide methods to deal with 'exceptional' situations
- Gives the programmer the option to keep the user from having the program stop without warning
- Again, this is not about fundamental CS, but about doing a better job as a programmer



# What counts as exceptional

- **errors.** indexing past the end of a list, trying to open a nonexistent file, fetching a nonexistent key from a dictionary, etc.
- **events.** search algorithm doesn't find a value (not really an error), mail message arrives, queue event occurs

## exceptions (2)

- ending conditions. File should be closed at the end of processing, list should be sorted after being filled
- weird stuff. For rare events, keep from clogging your code with lots of if statements.

# Error Names

Errors have specific names, and Python shows them to us all the time.

```
>>> input_file = open("no_such_file.txt", 'r')
Traceback (most recent call last):
  File "<pyshell#0>", line 1, in <module>
    input_file = open("no_such_file.txt", 'r')
IOError: [Errno 2] No such file or directory: 'no_such_file.txt'
>>> my_int = int('a string')
Traceback (most recent call last):
  File "<pyshell#1>", line 1, in <module>
    my_int = int('a string')
ValueError: invalid literal for int() with base 10: 'a string'
>>>
```

You can recreate an error to find the correct name. Spelling counts!

# a kind of non-local control

Basic idea:

- keep watch on a particular section of code
- if we get an exception, raise/throw that exception (let it be known)
- look for a catcher that can handle that kind of exception
- if found, handle it, otherwise let Python handle it (which usually halts the program)

# Doing better with input

- In general, we have assumed that the input we receive is correct (from a file, from the user).
- **This is almost never true.** There is always the chance that the input could be **wrong**
- Our programs should be able to handle this.

# Worse yet, input is evil

- "Writing Secure Code", by Howard and LeBlanc
  - "All input is evil until proven otherwise"
- Most security holes in programs are based on assumptions programmers make about input
- Secure programs protect themselves from evil input

# Rule 7

All input is evil, until proven otherwise

# General form, version 1

```
try:
```

```
    suite
```

```
except a_particular_error:
```

```
    suite
```



# try suite

- the `try` suite contains code that we want to monitor for errors during its execution.
- if an error occurs anywhere in that `try` suite, Python looks for a handler that can deal with the error.
- if no special handler exists, Python handles it, meaning the program halts and with an error message as we have seen so many times 😞

# except suite

- an `except` suite (perhaps multiple `except` suites) is associated with a `try` suite.
- each exception names a type of exception it is monitoring for.
- if the error that occurs in the `try` suite matches the type of exception, then that `except` suite is activated.

## try/except group

- if no exception in the `try` suite, skip all the `try/except` to the next line of code
- if an error occurs in a `try` suite, look for the right exception
- if found, run that `except` suite and then skip past the `try/except` group to the next line of code
- if no exception handling found, give the error to Python

## Code Listing 5.2

### Find a line in a file

```
1 # read a particular line from a file. User provides both the line
2 # number and the file name
3
4 file_str = input("Open what file:")
5 find_line_str = input("Which line (integer):")
6
7 try:
8     input_file = open(file_str)           # potential user error
9     find_line_int = int(find_line_str)    # potential user error
10    line_count_int = 1
11    for line_str in input_file:
12        if line_count_int == find_line_int:
13            print("Line {} of file {} is {}".format(find_line_int, file_str,
14            line_str))
15            break
16            line_count_int += 1
17    else:
18        # get here if line sought doesn't exist
19        print("Line {} of file {} not found".format(find_line_int, file_str))
20    input_file.close()
```

# Reminder, rules so far

1. Think before you program!
2. A program is a human-readable essay on problem solving that also happens to execute on a computer.
3. The best way to improve your programming and problem solving skills is to practice!
4. A foolish consistency is the hobgoblin of little minds
5. Test your code, often and thoroughly
6. If it was hard to write, it is probably hard to read. Add a comment.
7. All input is evil, unless proven otherwise.