Chapter 8 — Files, Exceptions & Debugging

This chapter introduces three closely connected skills: working with text files, handling expected run-time problems with exceptions, and systematically finding and fixing defects.

These skills allow Python programs to work with persistent data and behave more reliably when something unexpected happens.

This chapter is based on the Week 8 lecture materials and the accompanying sections on Debugging, Errors and Exceptions, and Files. The chapter follows their concepts and examples while reorganising them into a student-friendly learning experience.

Learning Outcomes

By the end of this chapter, you should be able to:

- Explain a text file as a sequence of lines and understand the role of the newline character \n.
- Open files for reading, writing, appending, or creation using open().
- Read a whole file, individual lines, or the remaining lines.
- Process a file line by line using a for loop.
- Write text to a file and close files correctly.
- Use a with statement to manage a file automatically.
- Count, search, filter, and process text stored in files.
- Recognise common error categories: syntax, logical/semantic, and run-time errors.
- Use try/except to handle expected exceptions, especially file-related problems.
- Distinguish between try, except, else, and finally.
- Recognise common built-in exceptions such as ValueError, NameError, ZeroDivisionError, FileNotFoundError, and FileExistsError.
- Apply systematic debugging strategies and use breakpoints and variable inspection in PyCharm.

Files: Why Programs Use Them

Variables hold data while a program is running. A file allows data to be stored outside the program so it can be read or processed later. Python is frequently used for processing text files.

A text file can be thought of as a sequence of lines. Each line ends with a newline character, \n, which marks the end of that line. When Python reads a line, that newline may be included in the string.

```
line = "Hello Python\n"
print(line)
```

Key idea: Think of a text file as a sequence of strings, one string for each line. This makes for loops useful for file processing.

File Handles and open()

When Python works with a file, information about the file is stored in a file object. A variable that refers to this object is called a file handle.

The basic form is:

```
f = open(filename, mode)
```

The filename is a string. The mode is optional and defaults to 'r'.

Mode	Purpose	Important behaviour
'r'	Read	Read an existing text file; this is the default mode.
'w'	Write	Write to a file; existing contents are erased. If the file does not exist, it is created.
'a'	Append	Write at the end of the file; if it does not exist, it is created.
'x'	Create	Create a new file.

Opening a file

```
f = open("mbox.txt", "r")
```

The returned file object is stored in f. You can then use methods of that object to read or write data.

File Names and Paths

A file can be in the current working directory or in another folder. When writing a Windows path, backslashes need attention because they can be interpreted as escape characters.

```
f = open("c:\\user\\john\\testfile.txt", "w")

# or use a raw string
f = open(r"c:\user\john\testfile.txt", "w")
```

Practical advice: For beginner programs, keeping the input/output files in the same project folder often makes file paths easier to manage.

Reading from a File

We introduce three important reading methods:

Method	Returns	Use when...
f.read()	The entire file as one string	You want all contents at once.
f.readline()	The next line as a string	You want to control line-by-line reading.
f.readlines()	The remaining lines as a list of strings	You want the remaining lines as a list.

The file pointer starts at the beginning. Reading moves the pointer forward. Once the contents have been read, the pointer is at the end. The seek() method can be used to move the pointer.

```
f = open("mbox.txt", "r")

text = f.read()
print(text)

f.close()
```

Reading Line by Line with a for Loop

A file object can be treated as a sequence of lines. This means a for loop can process each line directly.

```
f = open("mbox.txt", "r")

for line in f:
    print(line)

f.close()
```

Why this is useful: For large text files, processing one line at a time gives you a simple way to search, count, filter, or transform the contents without first turning the whole file into one string.

Closing Files and Using with

Once a file is no longer needed, it should be closed. Closing releases resources associated with the file.

```
f = open("mbox.txt", "r")
print(f.read())
f.close()
```

A with statement provides a convenient way to manage a file. When the with block is finished, the file is closed automatically.

```
with open("mbox.txt", "r") as f:
    print(f.read())
```

Good habit: For new code, prefer with open(...) as f: when you can. It keeps the file-management logic simple and ensures the file is closed when the block is exited.

Common File-Processing Patterns

Count lines

```
f = open("mbox.txt", "r")
count = 0

for line in f:
    count = count + 1

f.close()
print("Line Count:", count)
```

The counter increases once for every line processed.

Search for lines beginning with text

```
with open("mbox-short.txt", "r") as f:
    for line in f:
        if line.startswith("From:"):
            print(line)
```

startswith() tests whether a string begins with the specified text.

Search for text anywhere in a line

```
with open("mbox-short.txt", "r") as f:
    for line in f:
        line = line.rstrip()
        if not "@uct.ac.za" in line:
            continue
        print(line)
```

The in operator checks whether a string occurs anywhere inside another string. rstrip() removes trailing whitespace, including the newline at the end of a line.

Writing to Files

Open a file in write ('w') or append ('a') mode before writing. The write() method writes a string; writelines() writes multiple strings from a sequence.

```
with open("output.txt", "w") as f:
    f.write("First line\n")
    f.write("Second line\n")
```

```
lines = ["First line\n", "Second line\n"]

with open("output.txt", "w") as f:
    f.writelines(lines)
```

Important: write() does not automatically add a newline. If you want separate lines, include \n in the strings you write.

Write vs append

- 'w' starts a new file contents state by clearing existing contents.
- 'a' keeps the existing contents and writes new text at the end.
- Choose the mode according to whether you want to replace or add to the file.

Worked Example: Count Matching Lines

Suppose we want to count the number of lines in a file that contain the text '@uct.ac.za'.

```
count = 0

with open("mbox-short.txt", "r") as f:
    for line in f:
        if "@uct.ac.za" in line:
            count = count + 1

print("Matching lines:", count)
```

1. Open the file for reading.
2. Start the counter at 0.
3. Process each line with a for loop.
4. Test whether the required text is in the line.
5. Increase the counter when the condition is True.
6. Print the final count after the file has been processed.

When a File Is Missing

Trying to open a file that does not exist can produce a FileNotFoundError.

```
f = open("stuff.txt")
```

The program stops with an exception unless the problem is handled. This is a good example of an unexpected external situation: the program cannot assume that every requested file exists.

Understanding Errors and Exceptions

There are three broad categories of errors.

Category	Meaning	Example
Syntax error	The program cannot be parsed because the source-code arrangement is invalid.	Missing ':' or malformed syntax.
Logical / semantic error	The program runs but does not mean or do what the programmer intended.	Using the wrong formula or condition.
Run-time error	A problem is detected while the program is running.	Missing file or invalid conversion.

Important distinction: A program can have correct syntax and still be wrong. Debugging therefore requires understanding what the program is intended to do and comparing that with what it actually does.

Defensive Programming and Programming by Contract

Defensive programming means writing programs that can cope with reasonably expected error situations. However, we also caution against cluttering clear code with unnecessary checks.

Programming by contract expresses what a function requires and what it guarantees using preconditions and postconditions.

```
def middleChar(s):
    """
    Returns the character at the middle of s.
    Precondition: len(s) > 0
    Postcondition: returns the character at the least position
                   closest to the middle of s.
    """
    return s[len(s) // 2]
```

The precondition says what the caller should provide. If the caller violates the contract, the better response may be to fix the caller rather than adding unnecessary defensive checks inside the function.

Handling Exceptions with try / except

The try/except structure separates risky actions from recovery actions. If the code in try succeeds, the except block is skipped. If an exception occurs, control jumps to the except block.

```
try:
    risky_action()
except:
    recovery_action()
```

Example: handling a bad file name

```
fname = input("Enter the file name: ")

try:
    f = open(fname)
except:
    print("File cannot be opened")
    quit()

count = 0
for line in f:
    if line.startswith("Subject:"):
        count = count + 1

f.close()
print("There were", count, "subject lines in", fname)
```

Here, opening the file is the risky operation. If it fails, the program reports the problem and stops instead of crashing.

try / except with Type Conversion

```
astr = "Hello Bob"

try:
    istr = int(astr)
except:
    istr = -1

print("First", istr)

astr = "123"

try:
    istr = int(astr)
except:
    istr = -1
```

```
print("Second", istr)
```

The first conversion fails, so the except block runs and istr becomes -1. The second conversion succeeds, so the except block is skipped.

else and finally

In addition to try and except, Python provides else and finally to refine exception handling.

```
try:
    print("try something here")
except:
    print("this happens only if it fails")
else:
    print("this happens only if it succeeds")
finally:
    print("this happens no matter what")
```

Part	When it runs
try	Attempts the risky operation.
except	Runs when an exception occurs in try.
else	Runs when try succeeds.
finally	Runs regardless of whether an exception occurred.

Common Built-in Exception Types

Exception	Typical situation
ValueError	A value has the wrong form for an operation, such as an invalid integer conversion.
NameError	A name is used that has not been defined.
ZeroDivisionError	A division operation attempts to divide by zero.
FileNotFoundError	A requested file cannot be found.
FileExistsError	An operation requiring a new file encounters an existing file.

Catching a specific exception

```
try:
    print(x)
except NameError:
    print("Variable x is not defined")
except:
    print("Something else went wrong")
```

When the type of problem is known, naming the exception can make the handling clearer and more specific.

Do not overuse exceptions: We emphasise that exception handling should not replace simple selection when a problem can be straightforwardly checked. Use exceptions particularly for problems such as missing files or other run-time situations that are appropriate to recover from.

Debugging: Fixing Defects Systematically

Debugging is the process of fixing programs that do not do what was intended. We use the term defects to emphasise that these problems are introduced by programmers and can be reduced through careful design and testing.

Mindset: A defect is an opportunity to understand the program better. Changing code blindly may appear to fix one symptom while introducing another defect.

Understand before changing

Before changing a line, understand what the program is doing, what it should do, and where the behaviour first differs from the intention.

Reduce the problem

Large programs are difficult to reason about as a whole. Break the problem into smaller pieces so the faulty section can be isolated.

Test as you code

Do not write a large program and test only at the end. Instead: write a few statements, run them, test, understand, debug, and repeat.

```
Write a small part
      ↓
Run / test
      ↓
Understand the result
      ↓
Fix the defect
      ↓
Add the next small part
      ↓
Repeat
```

Peek inside with print()

When you cannot see what is happening inside a running program, temporary print statements can reveal whether a section was reached and what values variables contain.

```
print("Reached the loop")
print("value of total:", total)
print("value of count:", count)
```

Useful question: Do not ask only 'Where did the program crash?' Ask 'What value did each important variable have immediately before the wrong behaviour?'

Isolate with comments

If a completed program is difficult to isolate, temporarily remove or comment out a large section. If the problem disappears, slowly restore the code until the defect appears again.

Do not be too clever

Clear code is easier to debug and maintain. Prefer straightforward code that you can explain and inspect over clever code that is difficult to reason about.

Debugging in PyCharm

A source-level debugger lets you pause program execution and inspect the state of the program. The key concept is the breakpoint.

Breakpoints

A breakpoint is a point where you want the debugger to temporarily halt execution so you can inspect variables.

Starting a debugging session

1. Place a breakpoint beside the relevant line in the PyCharm editor.
2. Run the program using the Debug command rather than the normal Run command.
3. When execution pauses, inspect the variables currently in scope.
4. Single-step through the program to observe how execution changes.
5. Resume execution to the next breakpoint or to the end.
6. Use what you observe to locate and correct the defect.

The worked example uses a trip-length program and a breakpoint to reveal that a calculation is producing an unexpected value. The debugger displays variable types and values, helping the programmer compare actual state with expected state.

Worked Debugging Example

Consider a program intended to convert miles to kilometres and accumulate the total. The course example contains a defect in the conversion calculation:

```
MILES_PER_KM = 0.621371

def milesToKm(miles):
    km = miles // MILES_PER_KM
    return km
```

A quick test produces a suspicious total. Instead of guessing, use a breakpoint inside milesToKm() and inspect the values of miles and km. The actual variable values reveal whether the conversion is behaving as intended.

Debugging lesson: The debugger does not magically fix the program. It gives you evidence about the program's actual state so that you can identify the defect and make an informed correction.

Integrated Example: Safe File Processing

The following example combines the main Week 8 ideas: user input, try/except, file opening, line-by-line processing, string searching, counting, and with.

```
fname = input("Enter the file name: ")

try:
    with open(fname, "r") as f:
        count = 0

        for line in f:
            if "Subject:" in line:
                count = count + 1

        print("Subject lines:", count)
```

```
except FileNotFoundError:
    print("File cannot be opened.")
```

The risky operation is opening/reading the requested file. If the file is available, the program processes each line and counts matching lines. If the file is missing, FileNotFoundError is handled.

1. Ask the user for a file name.
2. Attempt to open the file inside try.
3. Automatically close the file with with.
4. Loop through each line.
5. Use in to select matching lines.
6. Update the counter.
7. Print the result.
8. If the file does not exist, execute the except block.

Applying Week 8 to the Mission Control Panel

The Mission Control Panel can now move beyond information stored only while the program is running. Week 8 provides the tools to make the system work with persistent mission records and recover gracefully from common file problems.

- Read mission logs from a text file.
- Count or search mission events line by line.
- Filter lines containing particular mission information.
- Write reports or processed results to an output file.
- Append new events to an existing mission log.
- Handle a missing or invalid file name without crashing.
- Use debugging techniques to inspect incorrect counters, conditions, and file-processing logic.

Design principle: Files provide persistent data; exceptions make file operations safer; debugging helps you understand and correct the program when its behaviour is not what you intended.

Checklist

Before moving on, make sure you can explain and use each item:

☐ What a file handle is.

☐ open(filename, mode) and the modes 'r', 'w', 'a', and 'x'.

☐ The difference between read(), readline(), and readlines().

☐ The file pointer and seek().

☐ Reading a file using for line in f.

☐ Why files should be closed.

☐ How with open(...) as f works.

☐ Counting lines in a file.

☐ Searching with startswith().

☐ Searching with in.

☐ Using rstrip() to remove trailing whitespace/newline characters.

☐ Writing with write() and writelines().

☐ The difference between write and append modes.

☐ What FileNotFoundError means.

☐ Syntax, logical/semantic, and run-time errors.

☐ Defensive programming.

☐ Preconditions and postconditions in programming by contract.

☐ The basic try/except structure.

☐ The roles of else and finally.

☐ Common built-in exceptions.

☐ Why exception handling should not be overused.

☐ What debugging means and why understanding the defect matters.

☐ Testing as you code.

☐ Using print statements to inspect execution and variable values.

☐ Isolating defects with comments.

☐ Why clear code is easier to debug.

☐ Breakpoints and variable inspection in PyCharm.

☐ Single-stepping and resuming execution in the debugger.

Practice Problems

1. Count lines: Write a program that asks for a file name and prints the number of lines in the file.

2. Search lines: Write a program that prints only lines beginning with 'From:'.

3. Filter text: Write a program that prints only lines containing a specified string.

4. Copy a file: Ask for a source file and a target file, then copy the source contents into the target.

5. Safe file opening: Modify a file-processing program so a missing file is handled using try/except.

6. Write a report: Read a file, count matching lines, and write the final count to a new output file.

7. Debug a counter: Insert temporary print statements to determine why a counter is producing an unexpected value.

8. Use the debugger: Set a breakpoint inside a loop and inspect how a variable changes on each iteration.

Summary

Week 8 connects Python programs to the outside world. Files let programs read and write persistent text data. File handles provide the interface to those files, while modes determine whether we read, replace, append, or create. Files can be processed efficiently line by line with for loops, and with statements provide convenient automatic closing.

Real programs also encounter problems. Understanding syntax, logical/semantic, and run-time errors helps you identify what kind of problem you are facing. try/except separates risky operations from recovery code, while else and finally provide additional control. At the same time, good programming practice does not mean adding checks everywhere: use defensive programming thoughtfully and make clear contracts for functions.

Finally, debugging is a systematic process of understanding what the program actually does, isolating the problem, testing small changes, and inspecting program state. Print statements and PyCharm breakpoints provide evidence that helps you find defects rather than guessing at fixes.

Appendix

Standard File Handles

Python's sys module defines three standard file handles:

Handle	Purpose	Need to open/close?
sys.stdin	Standard input, usually the keyboard	No
sys.stdout	Standard output, usually the console	No
sys.stderr	Standard error output, usually the console	No

Processing Text with split()

The Files section also demonstrates that string methods can be combined with file processing. For example, when a Python script contains import statements, split() can be used to separate the words on a line so that the imported module name can be extracted.

```
line = "from gert import harry"
parts = line.split()
print(parts)
```

This produces a list of separate words. The important Week 8 idea is that file processing and string processing work together: read a line, inspect or split its text, then extract the information needed.

Understanding Tracebacks

When a run-time error occurs, Python's traceback identifies useful information such as the file, line number, function involved, and exception type. Reading the traceback is part of debugging.

```
def middleChar(s):
    return s[len(s) // 2]

print(middleChar(None))
```

This produces a TypeError because None has no length. A related example using an empty string produces an IndexError because the calculated index is outside the string. These examples show why the same function can fail for different reasons depending on its input.

```
print(middleChar(""))   # IndexError: string index out of range
```

Defensive Programming: When Not to Add Checks

A programmer should handle reasonably expected external problems, but should not automatically add defensive checks for every possible misuse. For functions with clearly stated preconditions, callers are responsible for satisfying those conditions.

Unavoidable or Unexpected Run-time Problems

Some run-time problems depend on circumstances outside the program's control. For example, a file may not exist, disk space may run out, or a network connection may be lost. try/except provides a way to separate recovery code from the normal code that performs the main task.