

Fundamentos de Programação

2025-2026

Programming Fundamentals

Class #6 – Files & Exceptions

Overview

- Text files and binary files
- Operations on files
 - Opening, reading, writing, closing
- Filenames and paths
- I/O Streams
- Errors
- Exceptions
- Assertions



So far ...

- The programs we have seen so far are transient in the sense that they run for a short time, take input and produce output, but **when they end, everything disappears.**

Problem

- Problem: Student Grades Calculator
- Scenario:
 - You are building a program that calculates the average grade of a group of students
 - For example, students of Programming Fundamentals
- Challenge:
 - If the program asks the user to manually enter each student's name and grade every time it runs, it becomes tedious and error-prone
 - especially for large groups.
- Solution:
 - Store the student data in a file (e.g., grades.txt) so the program can read it automatically.
 - This way, the input is reusable, editable, and doesn't require retyping.

Possible solution (pseudo code)

```
SET total = 0
SET count = 0

OPEN the file "grades.txt" for reading

FOR each line in the file DO
    SPLIT the line into name and grade
    CONVERT grade to a number
    ADD grade to total
    INCREMENT count by 1
END FOR

IF count > 0 THEN
    SET average = total / count
    DISPLAY "Average grade is" + average
ELSE
    DISPLAY "No grades found."
END IF

CLOSE the file
```

Sample File Content:

```
Alice, 85
Bob, 78
Charlie, 92
```

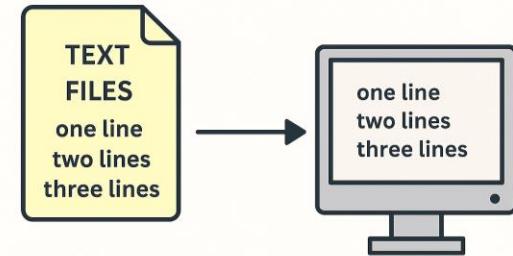
Key Reasons Files Matter

- Input and Output
 - Files serve as input sources or output destinations for automation.
 - Example: reading a list of tasks from a file, writing results to a report.
- Persistent Storage
 - Files let programs save data that survives after the program ends.
 - Example: saving user settings, game progress, or logs.
- Data Exchange
 - Programs can read from or write to files to share data with other systems.
 - Example: exporting results to CSV, importing configuration from JSON.
- Handling Large Data
 - Files allow access to large datasets without loading everything into memory.
 - Useful for processing logs, images, or databases.
- Backup and Archiving
 - Files help preserve historical data or snapshots.
 - Example: saving versions of a document or experiment results.
- Testing and Debugging
 - Files can store test cases, logs, or error traces for analysis.
 - Example: writing debug info to a log file during development.

Before we start our topic ...

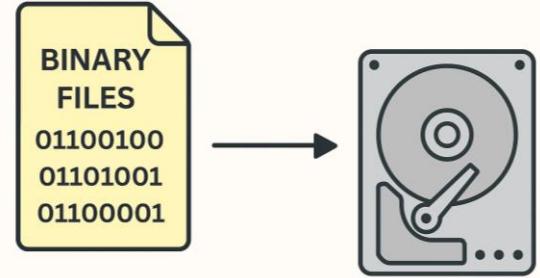
SOME USEFUL INFORMATION

Text Files



- Most of the programs we have seen so far are transient in the sense that they run for a short time, take input and produce output, but when they end, everything disappears.
- One of the simplest ways for programs to maintain their data is by reading and writing text files.
- **A text file is a sequence of characters stored on a persistent medium** like a hard drive, flash memory, or CD-ROM.
- **Characters are encoded in bytes** according to a standard coding table
 - such as ASCII, Latin-1 or UTF-8, for instance.

Binary files



- Like text files, binary files offer a way for programs to preserve data beyond their runtime.
- But instead of storing characters, **binary files contain raw byte sequences** that represent more complex data
 - such as images, audio, or structured records.
- These files are **not meant to be human-readable**
 - their contents depend on how the data was encoded and must be interpreted accordingly.
- Binary formats are often more compact and efficient than text, and they allow programs to store and retrieve data with exact fidelity, without relying on character encodings like ASCII or UTF-8.

Operations on files

- **Reading**
 - allows the program **to retrieve stored information**, whether it's configuration data, user input, or saved results.
- **Writing**
 - enables the program to record new information, update existing content, or generate output for future use.
- But to do that, **we need to have access to the file**.
- Access involves...
 - **establishing a connection between the program and the file system**
 - using a defined mode that indicates the intended operation
 - such as reading existing content, writing new data, or appending to what's already there.
- It's essential to manage this access responsibly: ensuring the file is properly opened, used, and then released when the task is complete.
 - This helps prevent data loss, corruption, or resource exhaustion.

FILES IN PYTHON



Opening files

- We must prepare a file before reading or writing.
 - This is called **opening the file**.
- The built-in function **open()** takes the name of the file and **returns a file object** that we use to access it.

```
fileobj = open(file_name, 'r') # open for reading  
fileobj = open(file_name, 'w') # open for writing
```

- More modes: 'a', 'r+', 'w+', 'a+', 'rb', ...

Text versus binary mode

- Normally, files are opened in **text mode**. This means:
 - You **write/read strings of characters** (type `str`).
 - Newline characters ('`\n`') are converted to/from platform-specific line endings:
 - LF in Unix, CRLF in Windows. ([About CRLF](#) in stackoverflow.)
 - Characters are encoded/decoded:
 - each character is converted to/from one or more bytes. (For example, '`á`' → 195, 161 in UTF-8).
 - You may specify the encoding with the optional `encoding=` argument.
`fileobj = open(file_name, 'r', encoding='utf-8')`
- For files that don't contain text, you should use '`wb`' or '`rb`' to open in **binary mode**. This means:
 - You **write/read strings of bytes** (type `bytes`, not `str`).
 - No conversions occur.



Reading a file

- We can use a for loop to **read a file line by line**.

```
fin = open('words.txt')

for line in fin:      # for each line from the file
    print(repr(line)) # do something with it

fin.close()
```

- Another way is using the **readline()** method.

```
while True:

    line = fin.readline()      # returns line to the end
    if line == "": break       # empty means end-of-file

    print(repr(line))
```

Reading a file (cont.)

- We can also **read the entire file** as string.

```
text = fin.read()  
# read as much as possible (up to EOF)
```

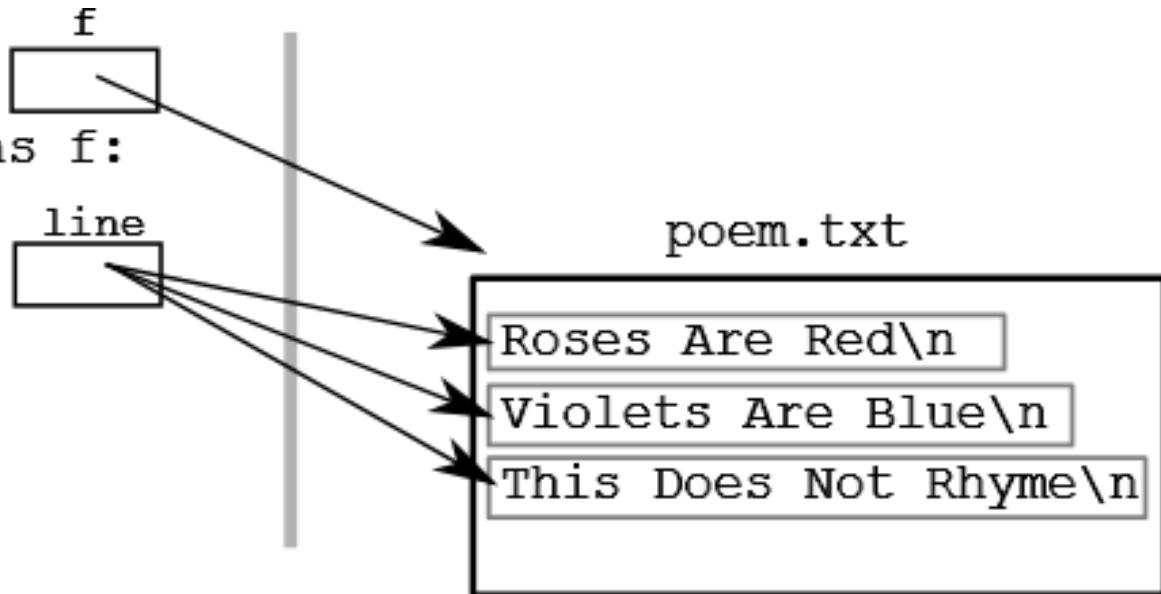
- Or read at most N characters.

```
str = fin.read(10)  
# read up to 10 chars (empty means EOF)
```



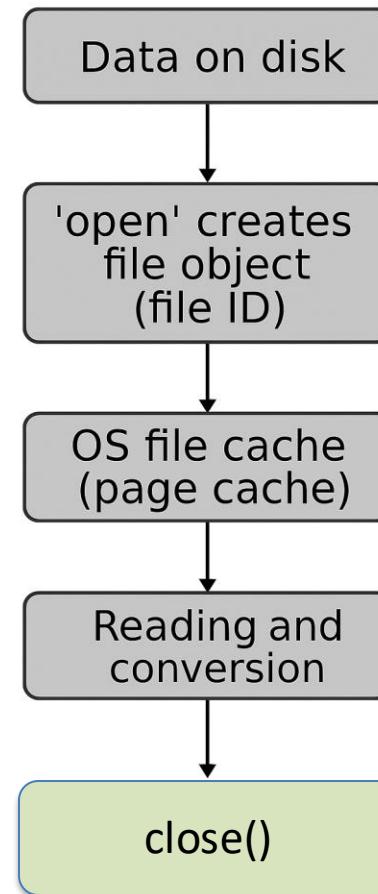
Reading file line by line

```
with open(filename) as f:  
    for line in f:  
        # Use line
```



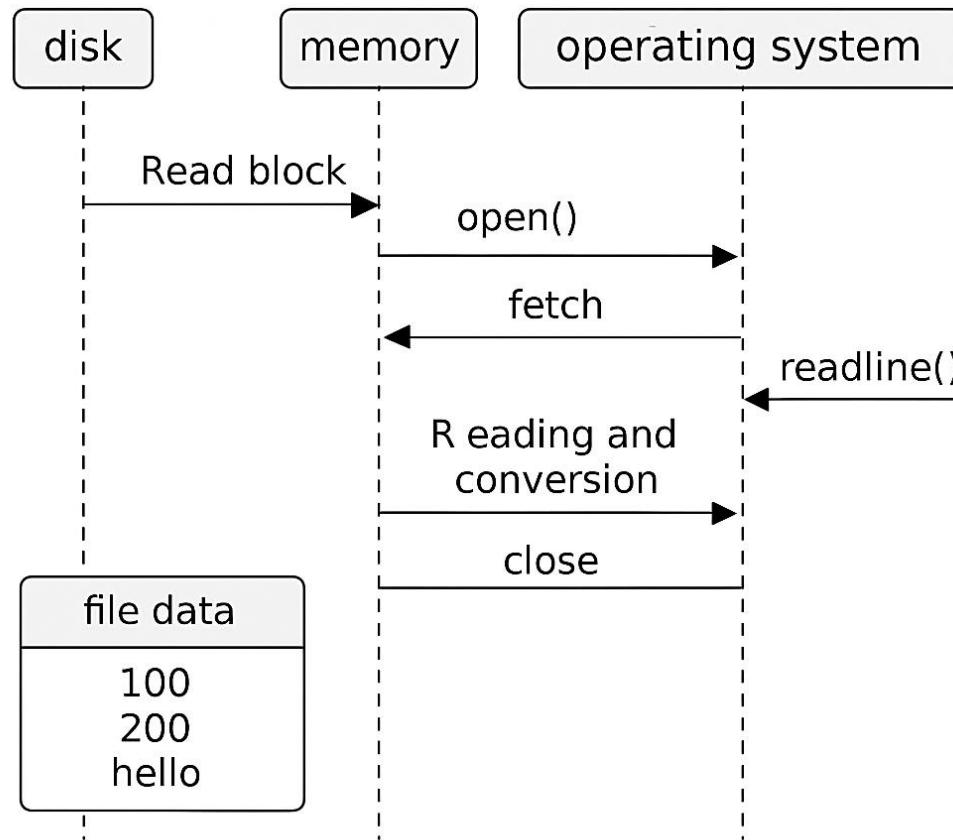
Python file-reading flow

- Disk
 - File stored as bytes.
- **open()**
 - Creates file object with ID and metadata.
- OS Cache
 - For faster access.
- **read() / readline()**
 - Reads and decodes bytes to strings.
- **close()**
 - Closes the file



In more detail - Sequence

Reading Values from a File



In more detail – File object

File Object (TextIOWrapper)

Class: io.TextIOWrapper

Inherits: io.IOBase → io.TextIOBase

Attributes / Components

- name: "data.txt"
- mode: "r"
- encoding: "UTF-8"
- errors: "strict"
- newline: None
- line_buffering: False
- buffer: BufferedReader
- closed: False

Methods

- read(), readline(), readlines()
- write(), writelines()
- seek(), tell()
- flush(), close()





Writing to a file (1)

- To write to a file, open it with mode '`w`' (or '`a`').

```
fout = open('output.txt', 'w', encoding='utf-8')
```

- '`w`' mode creates a new file or truncates an existing one
 - it deletes the old data and starts from scratch.
- '`a`' mode does not truncate; it appends to the end of the file.

- The `write()` method puts data into the file.

```
line1 = "To be or not to be,\n"
fout.write(line1)
```

- The file object keeps track of where it is; if you call `write` again, it adds the new data to the end.

```
line2 = "that is the question.\n"
fout.write(line2)
```

Writing to a file (2)

- The argument of write has to be a string, so we have to convert other types of values.

```
x = 0.75
```

```
fout.write('X: ' + str(x))
```

- Or use the string formating.

```
fout.write('{} costs {:.2f}€.'.format('tea', x))
```

```
fout.write(f'{tea} costs {x:.2f}€.')
```

- You may also use print with the file= argument.

```
print('X:', x, file=fout)
```

```
print('{} costs {:.2f}€.'.format('tea', x), file=fout)
```



Closing files

- After using the file, remember to close it.

```
fileobj.close()
```

- Better: use **with** statement.

- It automatically closes files.

```
with open(file_name, mode) as fileobj:  
    statements to read/write fileobj  
# fileobj.close() not required!
```

- **Guarantees closure**, even if an error occurs mid-operation

Why we need to use close() ?

- **Flushes Write Buffers**
 - If you're writing to a file, data may be held in memory (buffered) before being written to disk.
 - close() ensures all buffered data is flushed and safely stored.
- **Releases System Resources**
 - Every open file consumes OS-level resources (like file descriptors).
 - close() frees those resources, preventing leaks
 - especially important in loops or large programs.
- **Avoids File Locking Issues**
 - Some systems lock files while they're open.
 - Not closing can block other programs or users from accessing the file.
- **Ensures Data Integrity**
 - Closing a file confirms that all operations are complete and the file is in a consistent state.
- **Prevents Silent Errors**
 - Without closing, you might not notice if a write fails or if the file wasn't saved properly.

PYTHON – MORE INFORMATION RELATED TO FILES

Moving the file object's position

- We generally read and write sequentially, from start to end.
- But sometimes we need to "jump" around.
- The `tell()` method tells you the current position within the file.
- The `seek(offset)` method changes the current file position to offset bytes from the start.
 - An optional argument can specify a different reference point.

```
a0 = f.readline()      # read a line
pos = f.tell()          # store the current position
a1 = f.readline()      # read second line
f.seek(pos)             # return to stored position
a2 = f.readline()      # read second line again (a2==a1)
```



Filenames and paths

- Every file has a name, but to find it, we also need to know where it lives in the file system, its path.
- In computing, a **path** is a string that indicates where a file is located within a file system.
- **Module os** provides functions for working with files and directories (**os** stands for “operating system”).
`os.getcwd()` returns the name of the **current directory**.
Example: `os.getcwd() #-> '/home/jmr/FP'`
- An **absolute path** starts with / (the topmost directory).
- You may find the absolute path to a file:
`os.path.abspath('aula06/aula06.pdf')`
`#-> '/home/jmr/FP/aula06/aula06.pdf'`
- A **relative path** starts from the current directory.
`'aula06/aula06.pdf'`

Using Paths to Read and Write Files

- To access a file, Python needs to **know where it is.**
 - Paths tell Python the file's location

- Absolute path example

```
# Reads a file using its full location
with open("/home/at/Documents/data.txt", "r") as f:
    content = f.read()
```

- Starts from the root of the file system
- Works regardless of where your script runs

- Relative path example

```
# Writes to a file in a subfolder (of current dir)
with open("results/output.txt", "w") as f:
    f.write("Experiment complete.")
```

- Relative to the current working directory
- Easier to use in portable scripts and projects

File properties and listing directories

- There are functions to **check existence and type of files**.
 - `os.path.exists(path)` checks if a path exists.
 - Returns True if the path points to an existing **file or directory**.
 - `os.path.isdir(path)` checks if a path is a **directory**.
 - `os.path.isfile(path)` checks whether a path is a **regular file**.
- And a function to **get the contents of a directory**.
 - `os.listdir()` returns a list of the files (and other **directories**) in the given directory.

INPUT / OUTPUT, STREAMS AND FILES

I/O Streams and Files in Python

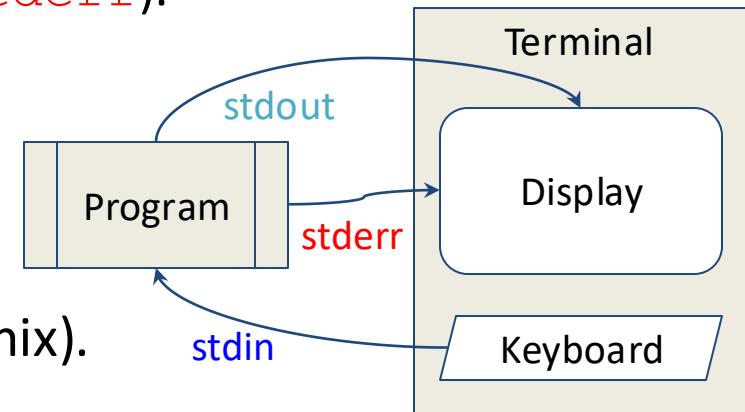
- An **I/O stream** is a **flow of data**, either into a program (input) or out of it (output).
 - Python uses streams to interact with files, the console, and other data sources.
- Files as Streams
 - When you open a file in Python, you create a stream between your program and the file.
 - This stream lets you read from or write to the file, one chunk at a time.

```
# Reading from a file (input stream)
with open("data.txt", "r") as f:
    content = f.read()

# Writing to a file (output stream)
with open("log.txt", "w") as f:
    f.write("Process started.")
```

Standard Streams

- Every program has **3 streams** connected to I/O devices in the computer:
 - A stream to get input text from the keyboard (`stdin`).
 - A stream to output text to the display (`stdout`).
 - A stream to output error messages (`stderr`).



- This is **standard in modern OS** (since Unix).
- In Python:

```
import sys  
sys.stdin  
sys.stdout  
sys.stderr
```

I/O redirection

- A user can redirect `stdout` to a file:

```
$ program > out.txt
```

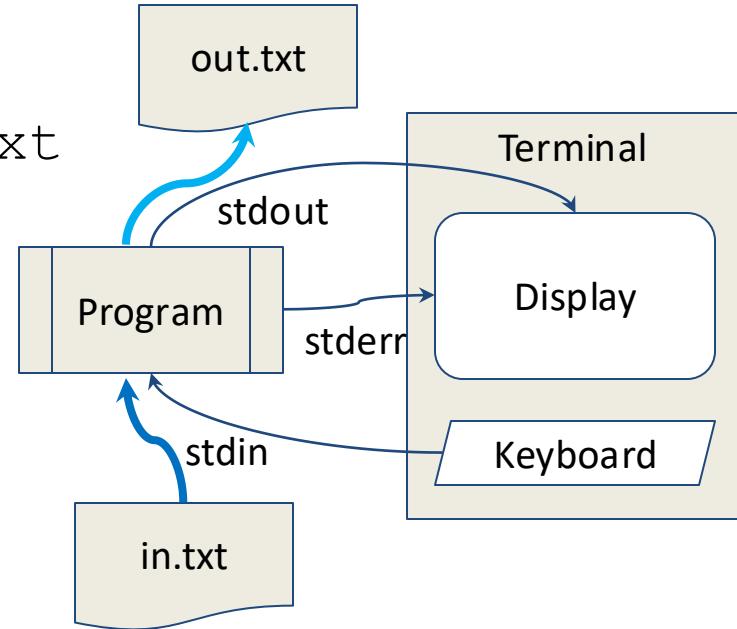
- He/She can also redirect `stdin` from a file:

```
$ program < in.txt
```

- Or redirect both streams:

```
$ program < in.txt > out.txt
```

- This is done by the shell.
 - Transparent to the program.
- Homework:
 - How to redirect `stderr` ?



Command Line Arguments

- Command-line arguments let us **pass information to a Python program when it starts**
 - like telling it which **file to open or where to save the output.**
- The `sys` module provides access to command-line arguments.
 - `sys.argv` is a variable with the **list of command-line arguments**
 - `len(sys.argv)` is the number of command-line arguments
 - `sys.argv[0]` is the **program (script) name.**

```
import sys
print('Number of args:', len(sys.argv))
print('Argument List:', sys.argv)
```

- Run above script as follows:

```
python3 test.py arg1 arg2 arg3
```

- Produces:

```
Number of arguments: 4 arguments.
```

```
Argument List: ['test.py', 'arg1', 'arg2', 'arg3']
```

- Module `getopt` can help parsing `sys.argv` [[Link](#)]

ERRORS

Many things can go wrong ...

- File Not Found (FileNotFoundException)
 - Trying to open a file that doesn't exist.
 - Example: `open("data.txt")` when "data.txt" is missing.
- Permission Denied (PermissionError)
 - Insufficient rights to read/write the file.
 - - Happens often with system files or protected directories.
- Wrong File Mode
 - Opening a file in read mode ('r') when it needs to be written ('w') or appended ('a').
 - Can cause FileNotFoundError or overwrite data unintentionally.
- Reading/Writing Mismatches
 - Reading from a binary file as if it were text, or vice versa.
 - Causes decoding errors like UnicodeDecodeError.
- Unexpected End of File
 - Reading more data than exists.
 - Can cause logic errors or crashes if not handled.
- File Corruption or Invalid Format
 - Trying to parse a malformed CSV, JSON, or XML file.
 - Raises ValueError, json.JSONDecodeError, etc.

What Can Go Wrong with User Input in Python

- **Wrong Data Type** (ValueError, TypeError)
 - Expecting a number but getting text.
 - Example: `int("hello")` → ValueError.
- **Encoding Issues**
 - Non-ASCII characters cause UnicodeEncodeError or UnicodeDecodeError.
- **Invalid Format**
 - Input doesn't match expected pattern (e.g., date, email).
 - Can cause parsing errors or logic failures.
- **Empty Input**
 - User presses Enter without typing anything.
 - May lead to unexpected behavior or crashes if not handled.
- **Unexpected Length**
 - Input too short or too long.
 - Can break string slicing or validation logic.

What Can Go Wrong with Calculations in Python

- Division by Zero (ZeroDivisionError)
 - Happens when dividing by 0 or 0.0.
 - Example: `x / 0` → crash unless handled.
- Invalid Type for Arithmetic (TypeError)
 - Trying to add a number to a string or list.
 - Example: `5 + "hello"` → TypeError.
- Bad Conversion (ValueError)
 - Converting a non-numeric string to a number.
 - Example: `int("abc")` or `float("NaN")`.
- Math Domain Errors (ValueError)
 - Using invalid inputs for math functions.
 - Example: `math.sqrt(-1)` or `math.log(0)`.
- Overflow in Integer Conversion (OverflowError)
 - Rare, but can occur with very large numbers in some contexts.
 - Example: `math.exp(1000)`.

EXCEPTIONS IN PYTHON

Exceptions

- Python provides an important feature to handle any unexpected events in your program: exceptions.
- You've seen exceptions before.

```
int("one") #-> ValueError: invalid literal for int()  
open("foo") #-> FileNotFoundError: No such file ...
```

- When Python encounters a situation that it cannot cope with, it **raises** an exception.
- That **interrupts the normal flow of execution**: the current function is interrupted, then the one that called it, etc., until the main program itself is interrupted.
- Information about the event is transmitted all the way through in an **exception object**.

Handling exceptions

- You can intercept selected exceptions and resume normal execution with the `try` statement.
- Example: handle errors accessing files:

```
try:  
    fh = open("testfile", "r")  
    content = fh.read()  
  
except IOError:  
    print("Error: could not open file or read data")  
  
else:  
    print("This executes iff no exception occurred")  
    fh.close()
```

- The `except` clause may name multiple exceptions.
- An `except` clause naming no exception, catches all types.

Exception information

- An exception can have an **argument**
 - which is a value that gives additional information about the problem.

```
def convert(var):  
    try:  
        return int(var)  
    except ValueError as e:  
        print("Not numeric:", e)  
    return None
```

```
m = convert("123")  
n = convert("xyz")
```

[Play](#)

Raising exceptions

- We can raise exceptions (of any type) by using the `raise` statement.

```
def checkLevel( level ):  
    if level < 1:  
        raise Exception(f"level={level} is too low!")  
    # code here is not executed if an exception is raised  
    return level  
  
try:  
    v = checkLevel(-1)  
    print("level = ", v)  
except Exception as e:  
    print("Error:", e)
```

[Play](#)



Assertions

- An **assertion** is a condition that the programmer knows (or believes) to be true at some point in a program.
- To check an assertion, we use `assert condition`.
- This evaluates the condition and, if false, raises an exception of type `AssertionError`.
- If that happens, the programmer learns that there is a **bug**.
 - He/she must find out why that assertion failed, and fix the problem.
- If users are confident that the program is correct, they can turn off assertion checking when running the program:
`python3 -O prog.py`.

Assertions: when to use?

- Assertions **at the start of a function** to check if arguments are within the domain of the function.
(Check preconditions.)
- Example:
 - To determine next day, parameters must define a valid date



```
def nextDay(y, m, d):
    assert dateIsValid(y, m, d)
    d += 1
    ...      # rest of solution here
    ...
    assert dateIsValid(y, m, d)
    return (y, m, d)
```

Assertions: when to use?

- At the end of a function, to check postconditions.

```
def nextDay(y, m, d):
    assert dateIsValid(y, m, d)
    d += 1
    ...      # rest of solution here
    ...
    assert dateIsValid(y, m, d) ←
    return (y, m, d)
```

- Assertions after calling functions for testing results.

```
def testNextDay():
    assert nextDay(1920, 2, 28) == (1920, 2, 29)
    assert nextDay(1920, 2, 29) == (1920, 3, 1)
    assert nextDay(1920, 12, 31) == (1921, 1, 1)
    print("ALL OK!")
```

SOME COMMON MISTAKES BEGINNERS MAKE WITH FILES IN PYTHON

Forgetting to close files manually

- Beginners often use `open()` without `close()`,
 - which can lead to resource leaks or locked files.
- Solution: use `with open(...)` as `f:` to handle this automatically.

Mistake

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

Fix

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

Using incorrect file paths

- Hardcoding paths like
"/wrong/path/data.txt"
 - without considering platform differences or relative paths.
- This causes FileNotFoundError.

```
# Mistake
with open("C:/wrong/path/data.txt", "r")
as f:
    content = f.read()
```

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

Overwriting files unintentionally

- Opening files in write mode ('w')
 - without realizing it erases existing content.
- Solution: Use 'a' for appending or check before writing.

```
# Mistake
with open("report.txt", "w") as f:
    f.write("New content")
```

```
# Fix
with open("report.txt", "a") as f:
    f.write("\nAdditional content")
```

Reading files without checking existence

- Trying to read a file that doesn't exist
 - without handling exceptions.
- Solution: Always wrap in try-except blocks for FileNotFoundError.

```
# Mistake
with open("data.txt", "r") as f:
    content = f.read() # Error if file is missing

# Fix
try:
    with open("data.txt", "r") as f:
        content = f.read()
except FileNotFoundError:
    print("The file does not exist.")
```

Misunderstanding encoding

- Ignoring encoding issues when reading non-ASCII files.
- Solution: Use `open(filename, encoding="utf-8")` to avoid `UnicodeDecodeError`.

```
# Mistake  
with open("unicode_file.txt", "r") as f:  
    text = f.read()
```

```
# Fix  
with open("unicode_file.txt", "r",  
          encoding="utf-8") as f:  
    text = f.read()
```

Reading entire files into memory

- Using `read()` on large files can crash the program.
- Solution: Prefer reading line by line
 - with `for line in f:`

```
# Mistake
with open("bigfile.txt", "r") as f:
    data = f.read()
```

```
# Fix
with open("bigfile.txt", "r") as f:
    for line in f:
        process(line)
```

Mixing file modes

- Trying to read from a file opened in write mode ('w') or vice versa.
- Solution: Always match the mode to the intended operation.

Mistake

```
with open("data.txt", "w") as f:  
    content = f.read()
```

Fix

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

Not testing for malformed files

- Beginners often assume files will always be present and correctly formatted.
- Solution: Include error-handling and test edge cases.

```
# Mistake
with open("data.txt", "r") as f:
    lines = f.readlines()
    number = int(lines[0]) # May crash if file is empty or malformed

# Fix
try:
    with open("data.txt", "r") as f:
        lines = f.readlines()
        if lines:
            number = int(lines[0])
        else:
            print("File is empty.")
except (FileNotFoundException, ValueError) as e:
    print("Problem reading file:", e)
```

**RETURNING TO OUR INITIAL
PROBLEM ...**

Possible Solution in Python

```
# Initialize total and count
total = 0
count = 0

# Open the file for reading
with open("grades.txt", "r") as file:
    for line in file:
        name, grade = line.strip().split(",")
        grade = float(grade)
        total += grade
        count += 1

# Calculate and display the average
if count > 0:
    average = total / count
    print("Average grade is:", average)
else:
    print("No grades found.")
```

Sample File Content:

```
Alice, 85
Bob, 78
Charlie, 92
```

More robust solution

```
# Initialize total and count
total = 0
count = 0
# Open the file for reading
with open("grades.txt", "r") as file:
    for line in file:
        parts = line.strip().split(",")
        if len(parts) != 2:
            continue # Skip lines that don't have exactly two parts

        name, grade_str = parts
        try:
            grade = float(grade_str)
            total += grade
            count += 1
        except ValueError:
            print(f"Skipping invalid grade for {name}: {grade_str}")

# Calculate and display the average
if count > 0:
    average = total / count
    print("Average grade is:", average)
else:
    print("No valid grades found.")
```