INT3075 Programming and Problem Solving for Mathematics

Files and Exceptions

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

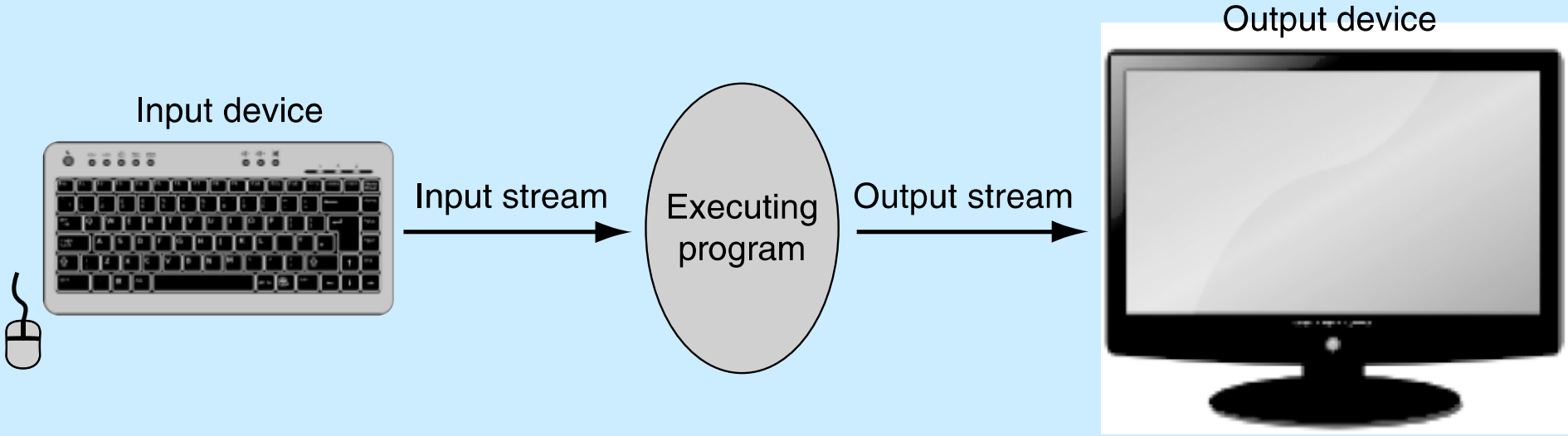
Two types of files

Files come in two general types:

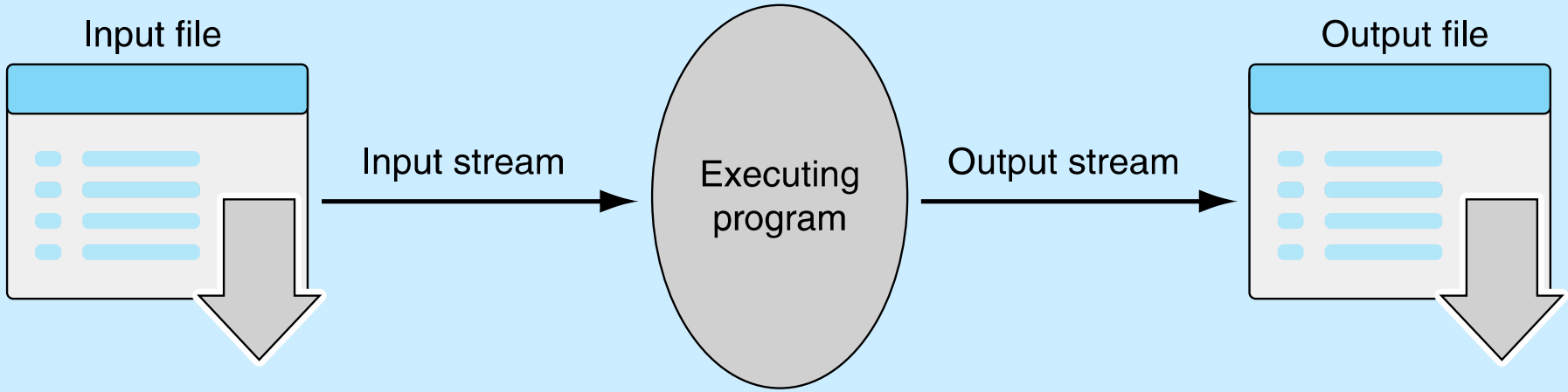
- *text files*. Files where control characters such as `"\n"` are translated. This are generally human readable
- *binary files*. All the information is taken directly without translation. Not readable and contains non-readable information

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.

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.

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:\gary\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 content 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 content

Text files use strings

- If you are interacting with text files (which is all we will do in this course), 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 content 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=temp_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()
```

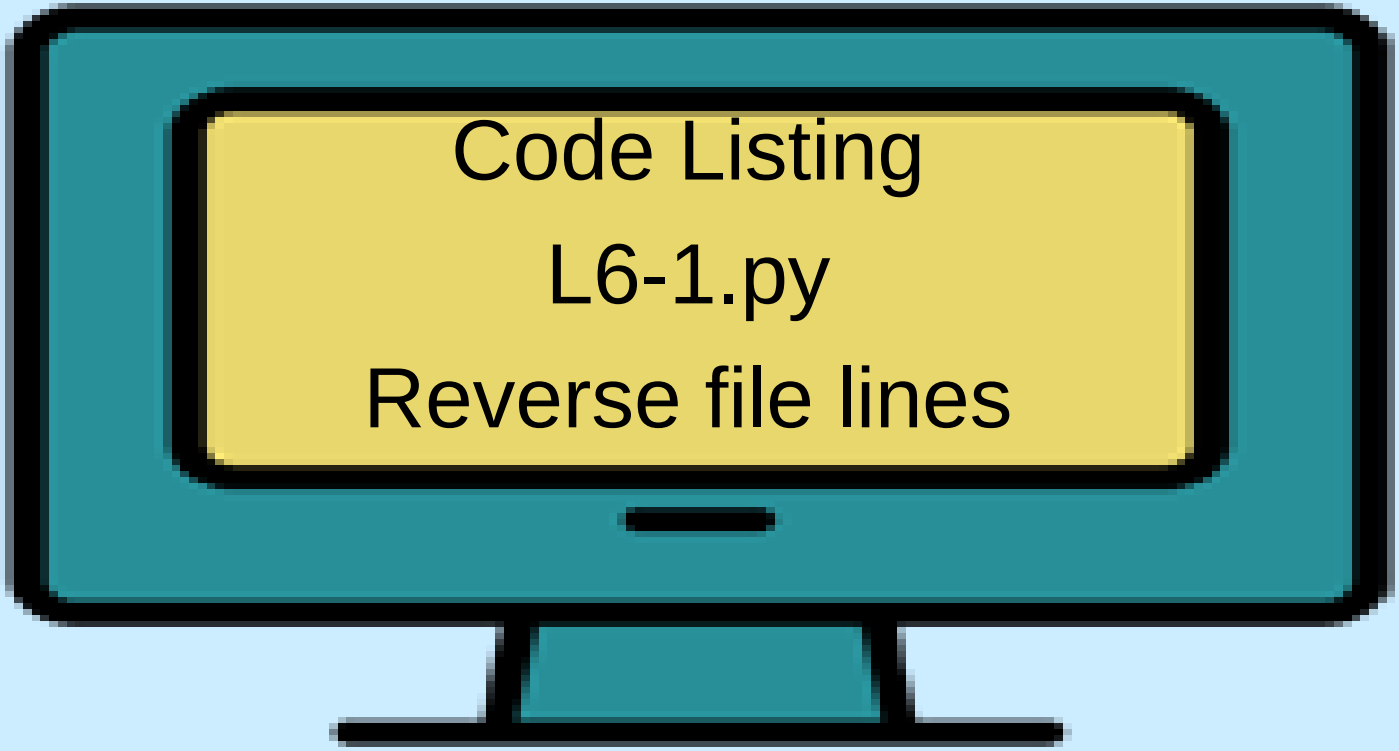
close

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 object

```
file_obj.close()
```

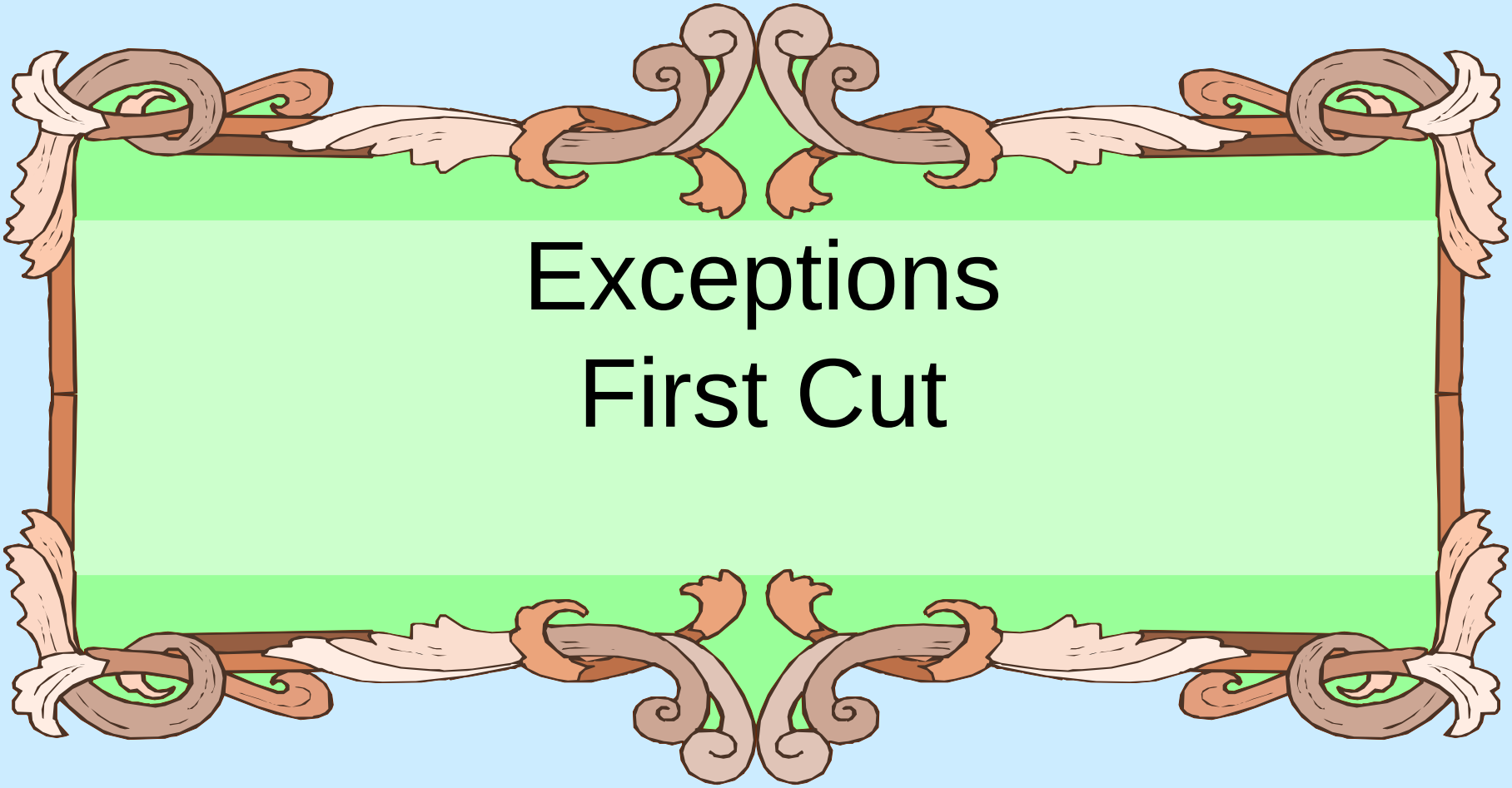
- All files should be closed!


Code Listing
L6-1.py
Reverse file lines

```
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()
```


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 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

- Most security holes in programs are based on assumptions programmers make about input
- Secure programs protect themselves from evil input

General form, version 1

```
try:
   suite
except a_particular_error:
   suite
```

try suite

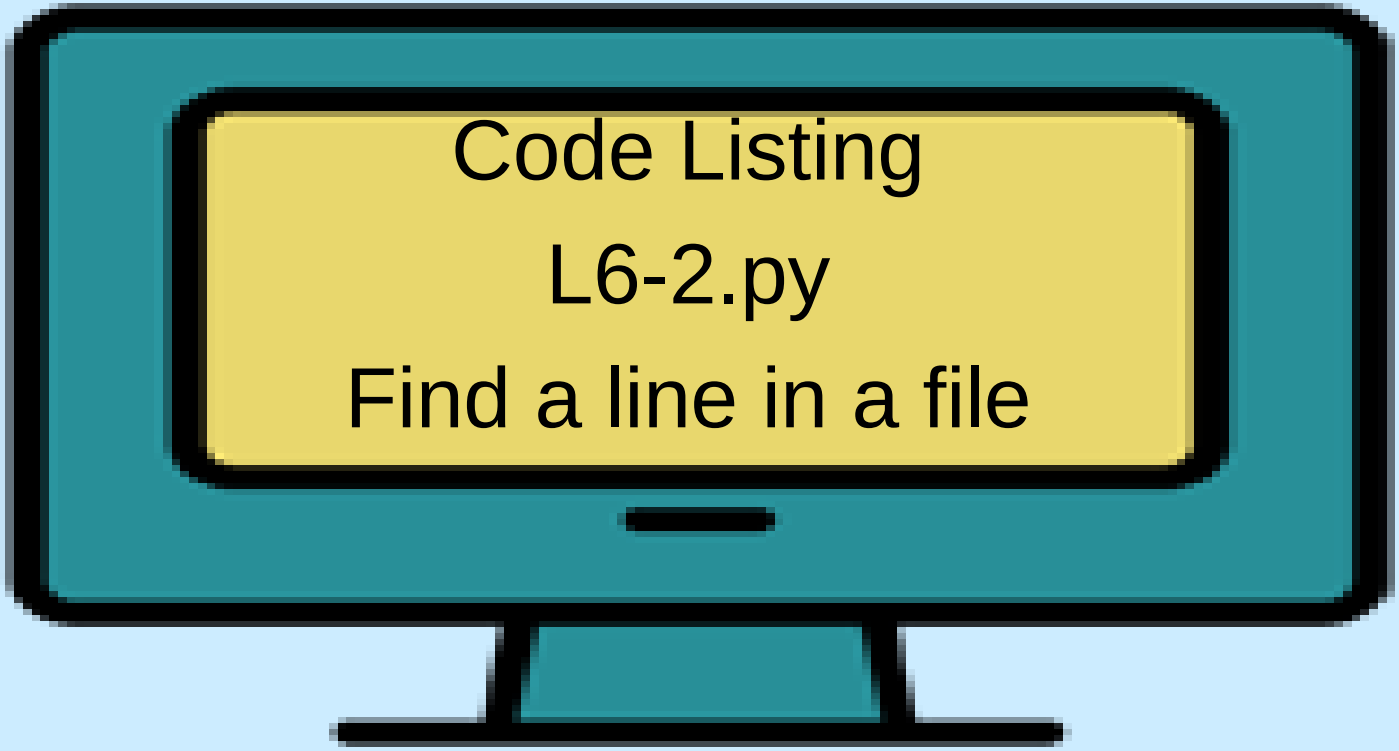
- the `try` suite contains code that we want to monitor for errors during its execution.
- if an error occurs anywhere in that `try` suite, Python looks for a handler that can deal with the error.
- if no special handler exists, Python handles it, meaning the program halts and with an error message as 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
L6-2.py
Find a line in a file

```
# read a particular line from a file. User provides both the line
# number and the file name

file_str = input("Open what file:")
find_line_str = input("Which line (integer):")

try:
    input_file = open(file_str)          # potential user error
    find_line_int = int(find_line_str)   # potential user error
    line_count_int = 1
    for line_str in input_file:
        if line_count_int == find_line_int:
            print("Line {} of file {} is {}".format(find_line_int, file_str, line_str))
            break
        line_count_int += 1
    else:
        # get here if line sought doesn't exist
        print("Line {} of file {} not found".format(find_line_int, file_str))
    input_file.close()

except FileNotFoundError:
    print("The file",file_str,"doesn't exist.")

except ValueError:
    print("Line",find_line_str,"isn't a legal line number.")

print("End of the program")

```