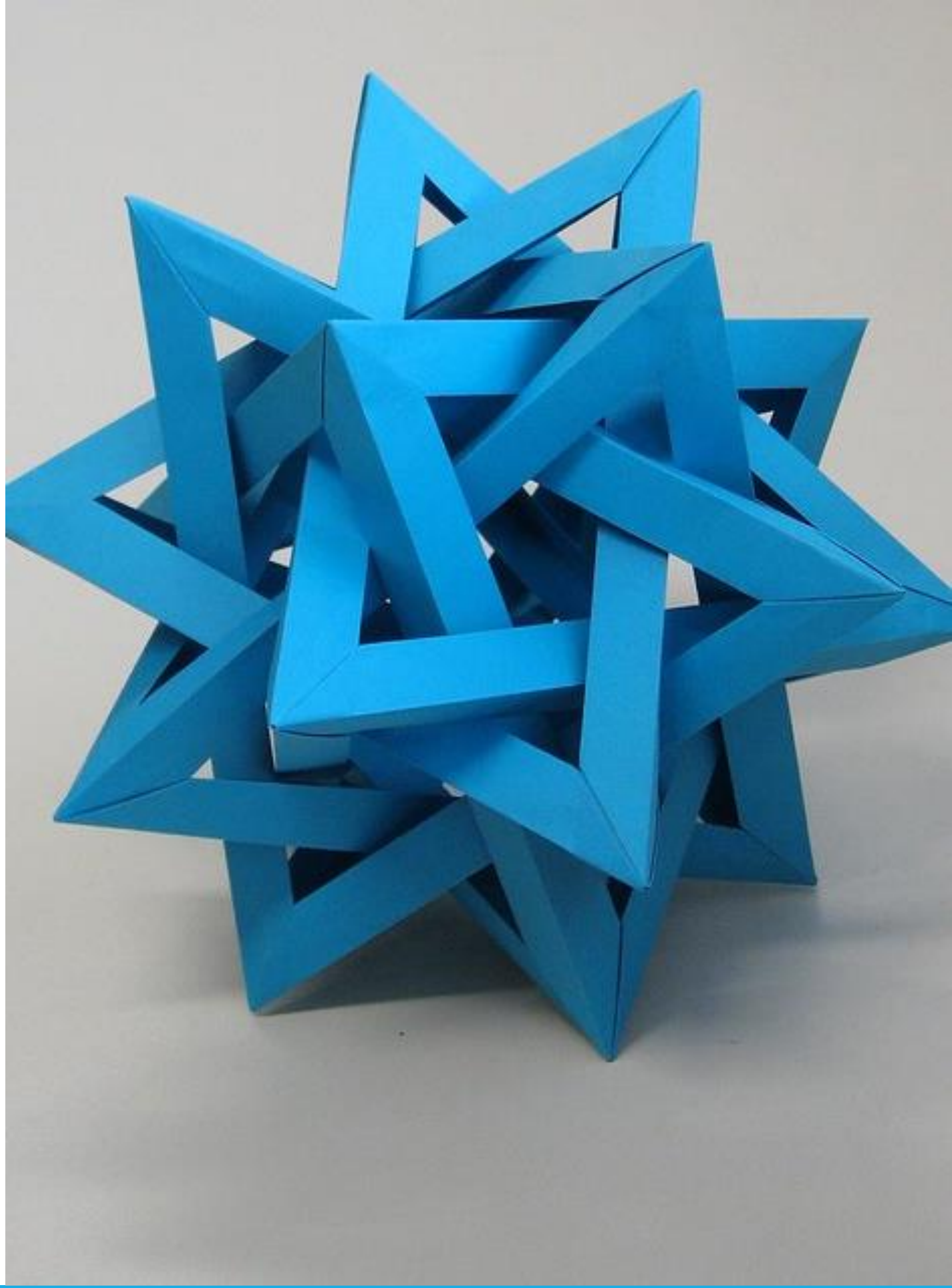
POLITECNICO DI TORINO

Dipartimento di Automatica e Informatica

Unit P7: Files and Error management

READING AND WRITING FILES, EXTRACTING FILE CONTENT, MANAGING ERRORS AND EXCEPTIONS

Chapter 7



Unit Goals

- To read and write text files
- To process collections of data
- To raise and handle exceptions

In this unit, you will learn how to write programs that manipulate files

Reading and Writing Text Files

📘 7.1

Reading and Writing Text Files

- Text files are very commonly used to store information
  - They are the most ‘portable’ types of data files
- Examples of text files include
  - files that are created with a simple text editor, such as Windows Notepad,
  - Python source code
  - HTML files
  - CSV files (comma-separated)
  - …

Opening Files: Reading

- To access a file, you must first open it
- Suppose you want to read data from a file named `input.txt`, located in the same directory as the program
- To open a file for reading, you must provide the name of the file as the first argument to the open function and the string `"r"` as the second argument:

```
infile = open("input.txt", "r")
```

- A “file object” is returned, that will be used for reading/writing

Opening Files: Reading (2)

- Important things to keep in mind:
  - When opening a file for reading, the file must exist (and otherwise be accessible) or an exception occurs
  - The file object returned by the open function must be saved in a variable
    - All operations for accessing a file are made via the file object

Opening Files: Writing

- To open a file for writing, you provide the name of the file as the first argument to the open function and the string "w" as the second argument:

```
outfile = open("output.txt", "w")
```

- If the output file already exists, **it is emptied** before the new data is written into it
- If the file does not exist, an empty file is created

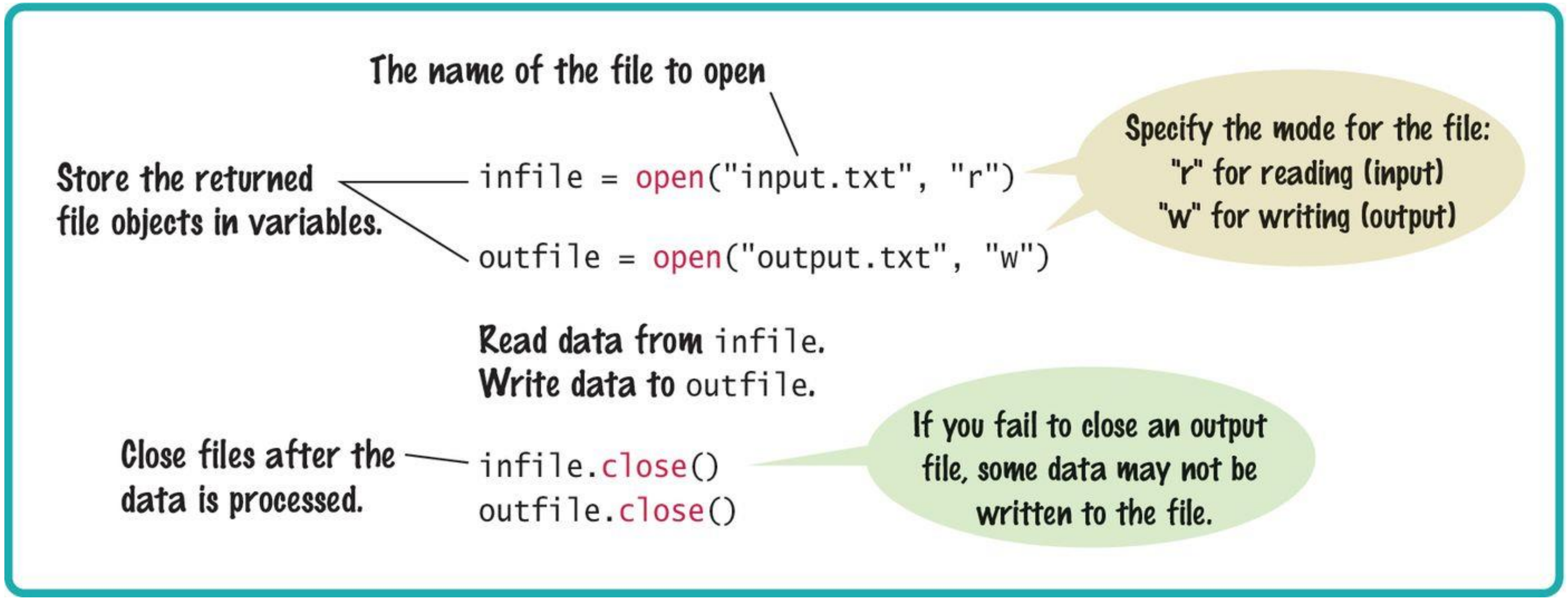
Closing Files: Important

- When you are done processing a file, be sure to close the file using the `close()` method:

```
infile.close()
outfile.close()
```

- If your program exits without closing a file that was opened for writing, some of the output may not be written to the disk file

Syntax: Opening And Closing Files



The name of the file to open

Store the returned file objects in variables.

```
infile = open("input.txt", "r")

outfile = open("output.txt", "w")
```

Specify the mode for the file:
"r" for reading (input)
"w" for writing (output)

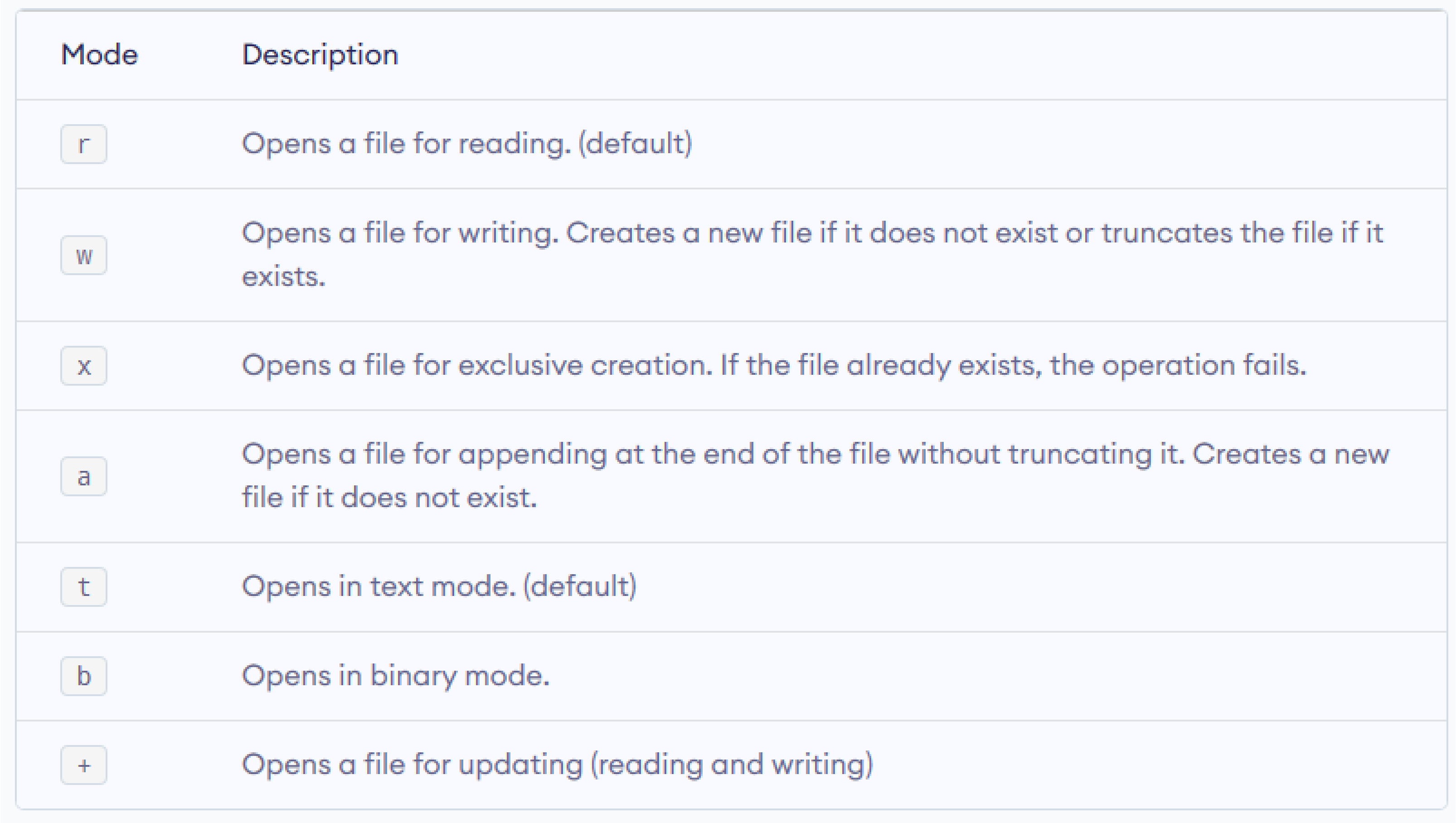
Read data from `infile`.
Write data to `outfile`.

Close files after the data is processed.

```
infile.close()
outfile.close()
```

If you fail to close an output file, some data may not be written to the file.

File Opening Modes



Mode	Description
`r`	Opens a file for reading. (default)
`w`	Opens a file for writing. Creates a new file if it does not exist or truncates the file if it exists.
`x`	Opens a file for exclusive creation. If the file already exists, the operation fails.
`a`	Opens a file for appending at the end of the file without truncating it. Creates a new file if it does not exist.
`t`	Opens in text mode. (default)
`b`	Opens in binary mode.
`+`	Opens a file for updating (reading and writing)

Reading From a File

- To read a line of text from a file, call the `readline()` method on the file object that was returned when you opened the file:

```
line = infile.readline()
```

- When a file is opened, an input marker (cursor) is positioned at the beginning of the file
- The `readline()` method reads the text, starting at the current position and continuing **until the end of the line is encountered**
  - The input marker is then moved to the next line

Reading From a File (2)

- For example, suppose `input.txt` contains the lines

  ```
  flying
  circus
  ```

- The first call to `readline()` returns the string `"flying\n"`
  - Recall that `\n` denotes the newline character that marks the end of the line
- If you call `readline()` a second time, it returns the string `"circus\n"`

Reading From a File (3)

- Calling `readline()` again yields the empty string `""` because you have reached the end of the file
- If the file contains a blank line, then `readline()` returns a string containing only the newline character `"\n"`

Reading **Multiple Lines** From a File

- You repeatedly read a line of text and process it until the sentinel value is reached
- The sentinel value is an empty string, which is returned by the `readline()` method after the end of file has been reached

```
line = infile.readline()
while line != "" :
    # Process the line.
    line = infile.readline()
```

Converting File Input

- As with the input function, the `readline()` method can only return strings
- If the file contains numerical data, the strings must be converted to the numerical value using the `int()` or `float()` function:

```
value = float(line)
```

- The newline character at the end of the line is ignored when the string is converted to a numerical value

Writing To A File

- For example, we can write the string "Hello, World!" to our output file using the `write()` method:

```
outfile.write("Hello, World!\n")
```

  - Unlike `print()`, when writing text to an output file, you must explicitly write the newline character to start a new line

- You can also write formatted strings to a file with the `write()` method:

```
outfile.write("Number of entries: %d\nTotal: %8.2f\n"
  % (count, total))
```

Example: File Reading/Writing

- Suppose you are given a text file that contains a sequence of floating-point values, stored one value per line
- You need to read the values and write them to a new output file, aligned in a column and followed by their total and average value
- If the input file has the contents

```
32.0
54.0
67.5
80.25
115.0
```

Example: File Reading/Writing (2)

- The output file will contain

```
32.00
54.00
67.50
80.25
115.00
--------
Total: 348.75
Average: 69.75
```

Example One

- Open the file total.py

Common Error

- Backslashes in File Names
  - When using a String literal for a file name with path information, you need to supply each backslash twice:

```
infile = open("c:\\homework\\input.txt", "r")
```

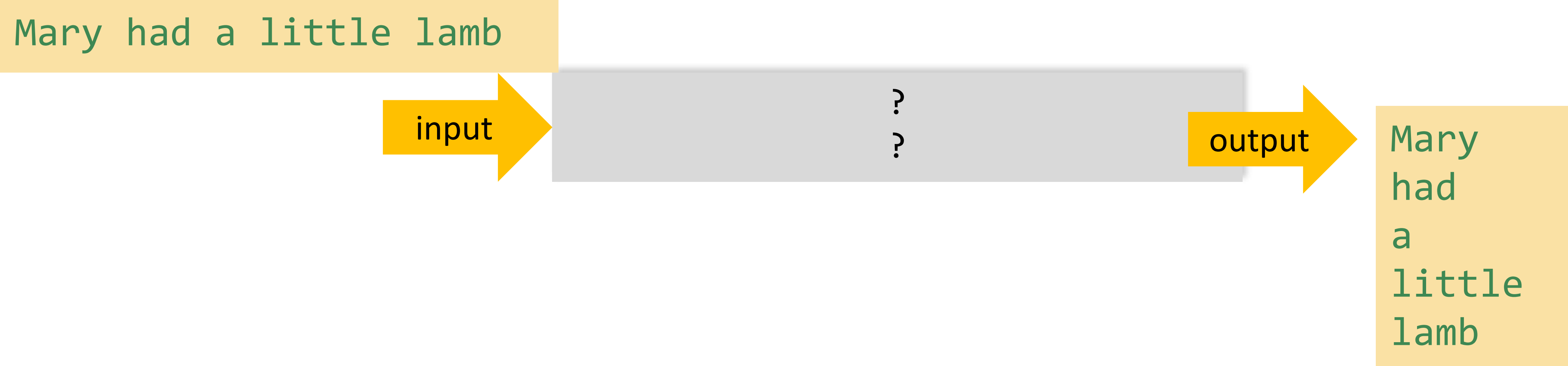
  - A single backslash inside a quoted string is the *escape character*, which means the next character is interpreted differently (for example, ‘\n’ for a newline character)
  - When a user supplies a filename into a program, the user should not type the backslash twice

Text Input and Output

📘 7.2

Text Input and Output

- How to process text with complex contents?
- How to cope with challenges that often occur with real data?
- Reading Words Example:

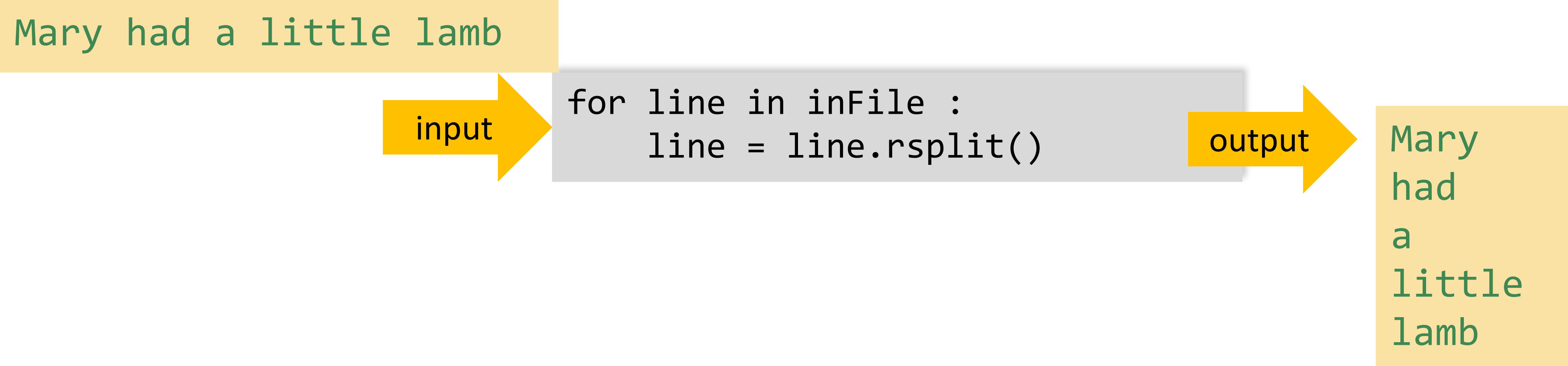


```
Mary had a little lamb
```

```
Mary
had
a
little
lamb
```

Text Input and Output

- How to process text with complex contents?
- How to cope with challenges that often occur with real data?
- Reading Words Example:



```
Mary had a little lamb
```

input

```
for line in inFile :
    line = line.rsplit()
```

output

```
Mary
had
a
little
lamb
```

Processing Text Input

- There are times when you want to read input by:
  - Each word
  - Each line
  - A single character

- Python provides methods such: `read()`, `split()` and `strip()` for these tasks

Processing text input is required for almost all types of programs that interact with the user

Text Input and Output

- Python can treat an input file as though it were a container of strings in which each line comprises an individual string
  - “Implicit readline()”
- For example, the following loop reads all lines and prints them:

```
for line in infile :
    print(line)
```

- At the beginning of each iteration, the loop variable `line` is assigned the value of a string that contains the next line of text in the file
- There is a critical difference between a file and a container:
  - Once you read the file you must close it before you can iterate over it again

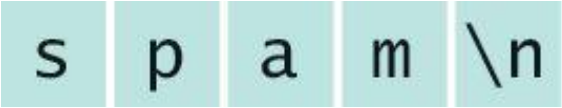
An Example of Reading a File

- We have a file that contains a collection of words; one per line:

```
spam
and
eggs
```

Removing The Newline (1)

- Recall that each input line ends with a newline (\n) character
- Generally, the newline character must be removed before the input string is used
- When the first line of the text file is read, the string line contains



s	p	a	m	\n

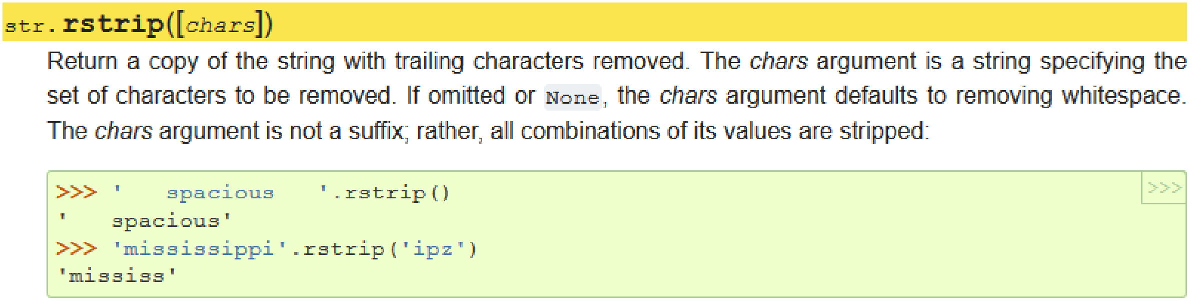
Removing The Newline (2)

- To remove the newline character, apply the `rstrip()` method to the string:

```
line = line.rstrip()
```

- This results in the string:

s	p	a	m



str.**rstrip**([*chars*])

Return a copy of the string with trailing characters removed. The *chars* argument is a string specifying the set of characters to be removed. If omitted or `None`, the *chars* argument defaults to removing whitespace. The *chars* argument is not a suffix; rather, all combinations of its values are stripped:

```
>>> '   spacious   '.rstrip()
'   spacious'
>>> 'mississippi'.rstrip('ipz')
'mississ'
```

https://docs.python.org/3/library/stdtypes.html#str.rstrip

Character Strip Methods

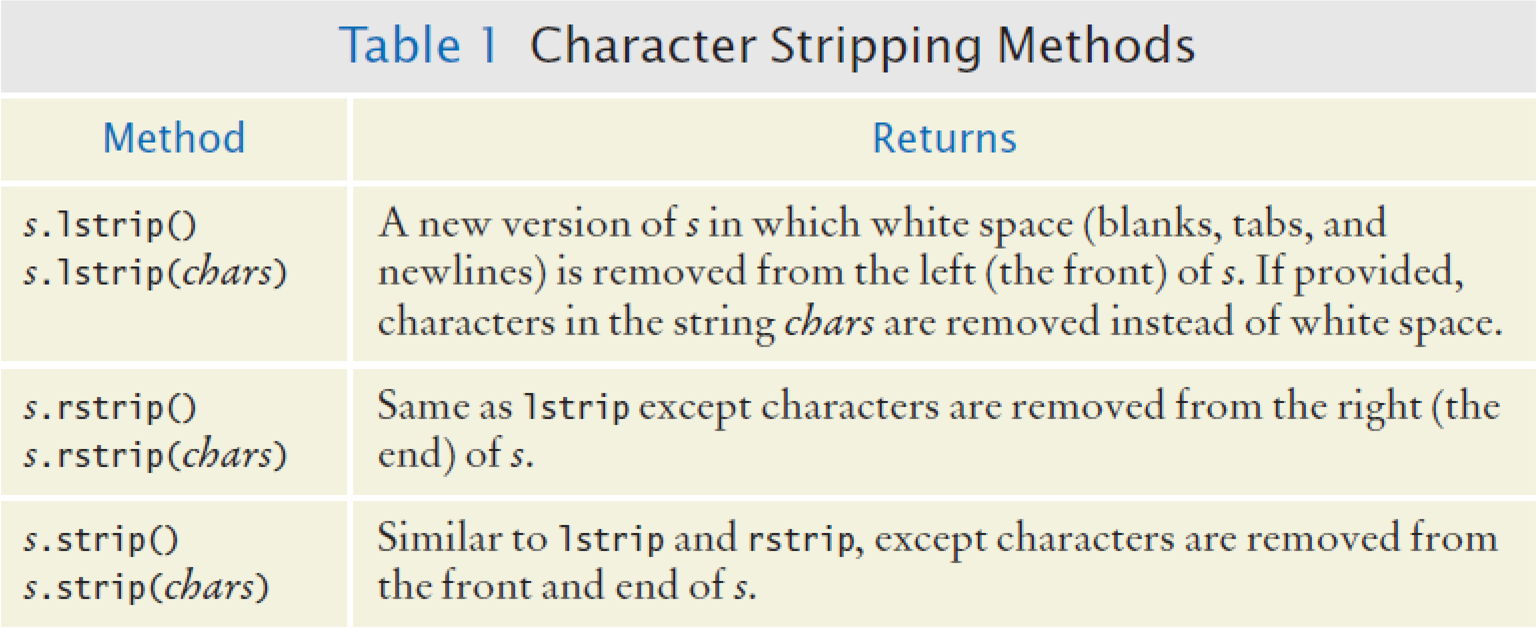


Table 1 Character Stripping Methods

Method	Returns
s.lstrip() *s*.lstrip(*chars*)	A new version of *s* in which white space (blanks, tabs, and newlines) is removed from the left (the front) of *s*. If provided, characters in the string *chars* are removed instead of white space.
s.rstrip() *s*.rstrip(*chars*)	Same as lstrip except characters are removed from the right (the end) of *s*.
s.strip() *s*.strip(*chars*)	Similar to lstrip and rstrip, except characters are removed from the front and end of *s*.

Character Strip Examples

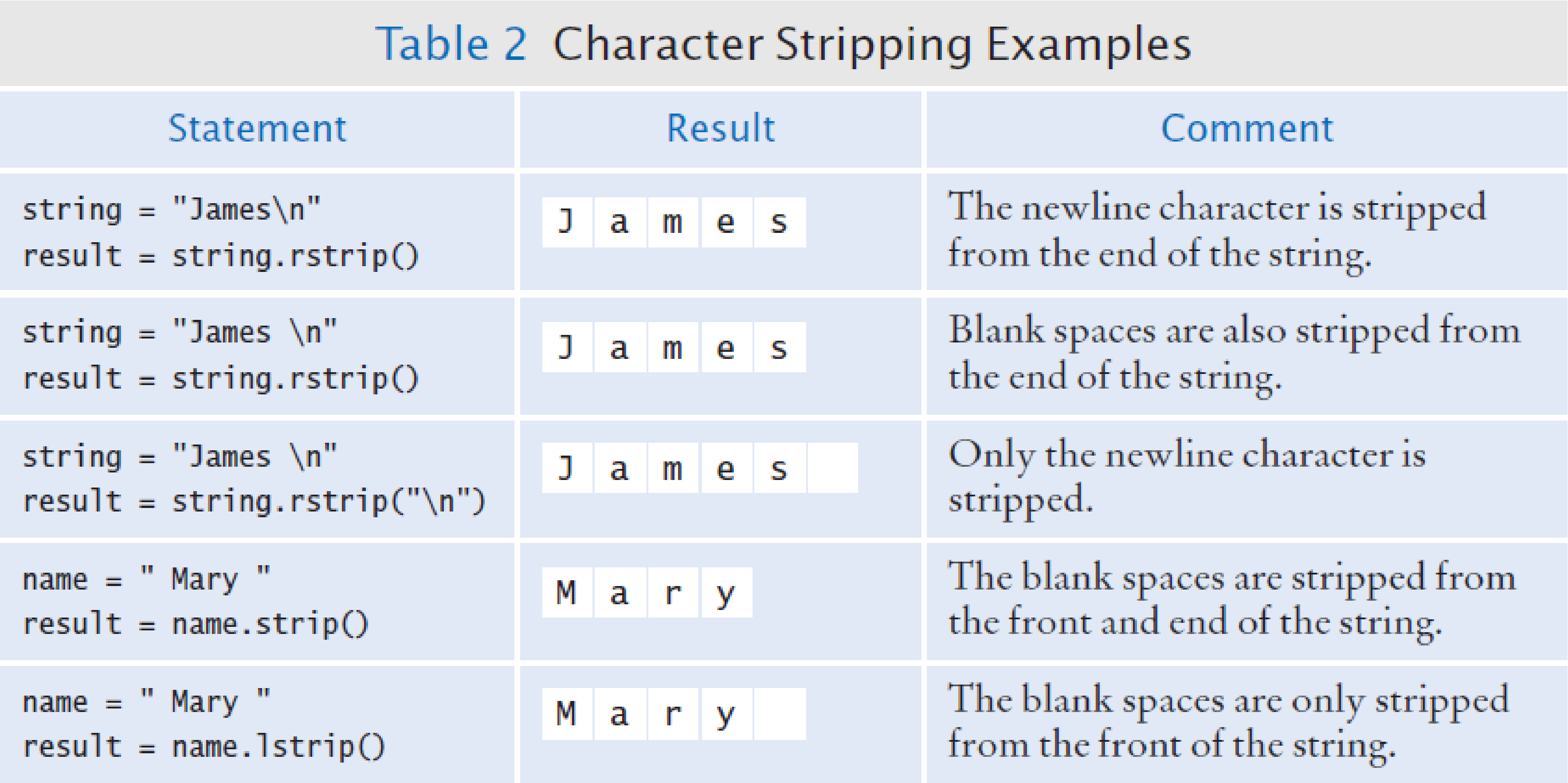


Table 2 Character Stripping Examples

Statement	Result	Comment

`string = "James\n"`
`result = string.rstrip()`
	`James`	The newline character is stripped from the end of the string.

`string = "James \n"`
`result = string.rstrip()`
	`James`	Blank spaces are also stripped from the end of the string.

`string = "James \n"`
`result = string.rstrip("\n")`
	`James `	Only the newline character is stripped.

`name = " Mary "`
`result = name.strip()`
	`Mary`	The blank spaces are stripped from the front and end of the string.

`name = " Mary "`
`result = name.lstrip()`
	`Mary `	The blank spaces are only stripped from the front of the string.

Reading Words

- Sometimes you may need to read the individual words from a text file
- For example, suppose our input file contains two lines of text

```
Mary had a little lamb,
whose fleece was white as snow
```

Reading Words (2)

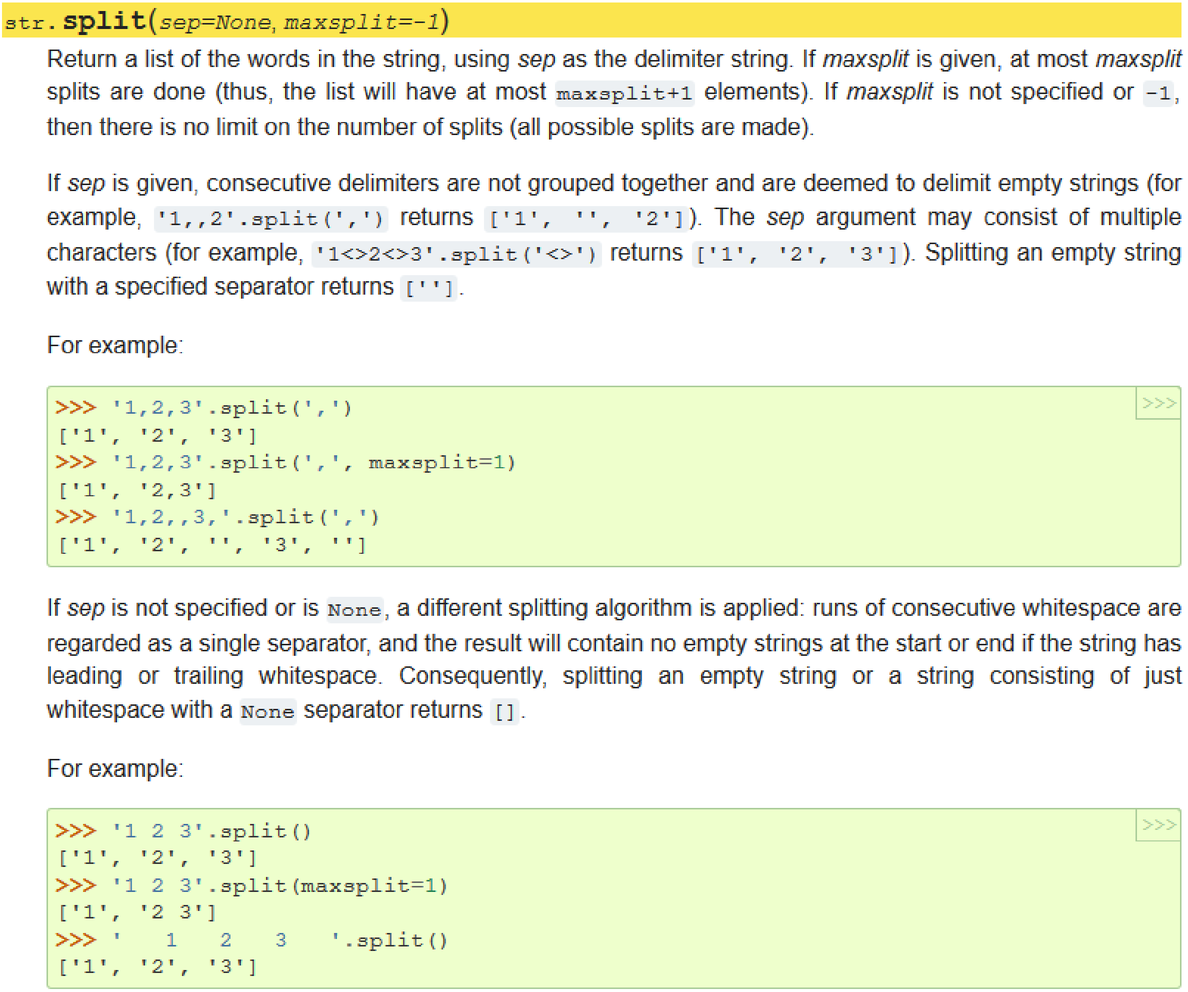
- We would like to print to the terminal, one word per line

```
Mary
had
a
little
. . .
```

- Because there is no method for reading a word from a file, you must first read a line and then split it into individual words

```
line = line.rstrip()
wordlist = line.split()
```

split()



str.**split**(*sep=None*, *maxsplit=-1*)

Return a list of the words in the string, using *sep* as the delimiter string. If *maxsplit* is given, at most *maxsplit* splits are done (thus, the list will have at most `maxsplit+1` elements). If *maxsplit* is not specified or `-1`, then there is no limit on the number of splits (all possible splits are made).

If *sep* is given, consecutive delimiters are not grouped together and are deemed to delimit empty strings (for example, `'1,,2'.split(',')` returns `['1', '', '2']`). The *sep* argument may consist of multiple characters (for example, `'1<>2<>3'.split('<>')` returns `['1', '2', '3']`). Splitting an empty string with a specified separator returns `['']`.

For example:

```
>>> '1,2,3'.split(',')
['1', '2', '3']
>>> '1,2,3'.split(',', maxsplit=1)
['1', '2,3']
>>> '1,2,,3,'.split(',')
['1', '2', '', '3', '']
```

If *sep* is not specified or is `None`, a different splitting algorithm is applied: runs of consecutive whitespace are regarded as a single separator, and the result will contain no empty strings at the start or end if the string has leading or trailing whitespace. Consequently, splitting an empty string or a string consisting of just whitespace with a `None` separator returns `[]`.

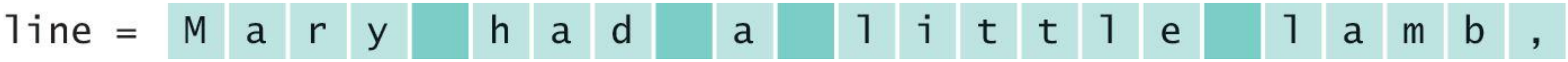
For example:

```
>>> '1 2 3'.split()
['1', '2', '3']
>>> '1 2 3'.split(maxsplit=1)
['1', '2 3']
>>> '   1   2   3   '.split()
['1', '2', '3']
```

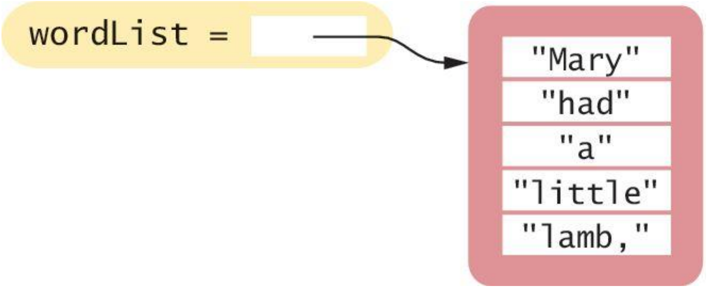
https://docs.python.org/3/library/stdtypes.html#str.split

Reading Words (3)

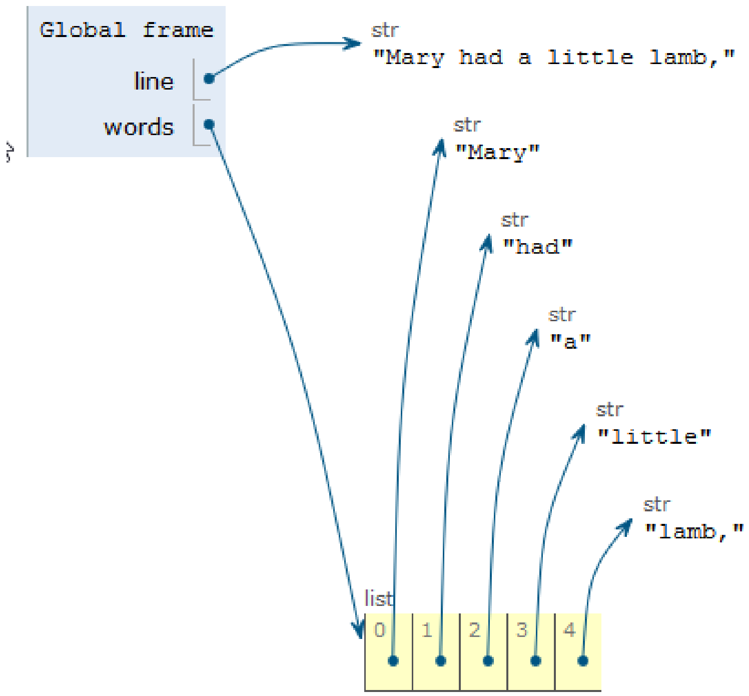
- The `split()` method returns the list of substrings that results from splitting the string at each blank space (or tabs or newlines)
- For example, if line contains the string:



- It will be split into 5 substrings that are stored in a list in the same order in which they occur in the string:

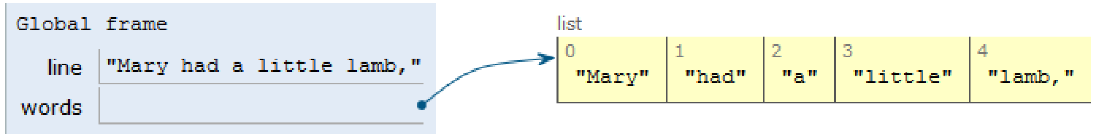


PythonTutor



```
line = 'Mary had a little lamb,'

words = line.split()
```



(different visualization options, it is actually the same value)

Reading Words (4)

- Notice that the last word in the line contains a comma
- If we only want to print the words contained in the file without punctuation marks, we can strip those from the substrings using the `rstrip()` method:

```
word = word.rstrip(".,?!")
```

Reading Words: Complete Example

```
inputFile = open("lyrics.txt", "r")
for line in inputFile :
    line = line.rstrip()
    wordList = line.split()
    for word in wordList :
        word = word.rstrip(".,?!")
        print(word)

inputFile.close()
```

Example Two

- Open the file lyrics.py

Additional String Splitting Methods

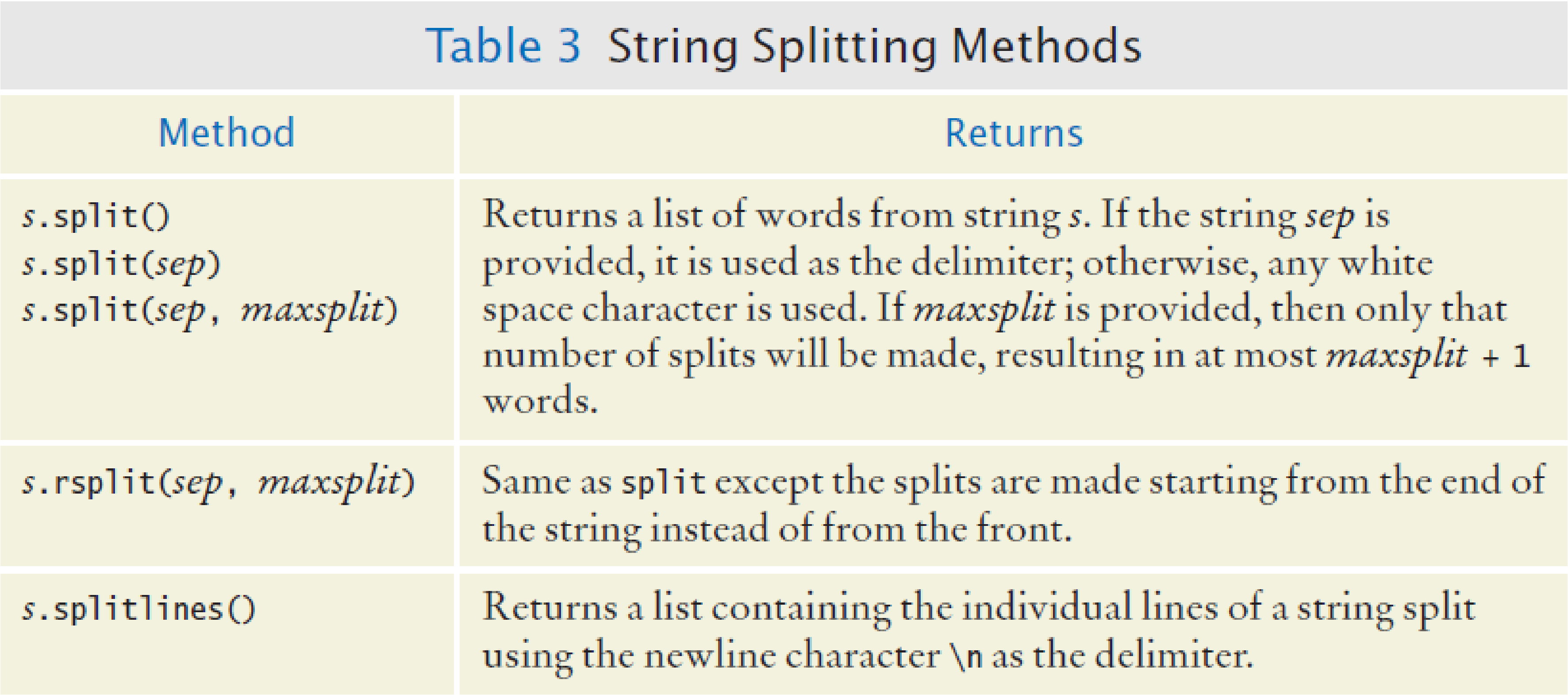


Table 3 String Splitting Methods

Method	Returns
s.split() *s*.split(*sep*) *s*.split(*sep*, *maxsplit*)	Returns a list of words from string *s*. If the string *sep* is provided, it is used as the delimiter; otherwise, any white space character is used. If *maxsplit* is provided, then only that number of splits will be made, resulting in at most *maxsplit* + 1 words.
s.rsplit(*sep*, *maxsplit*)	Same as split except the splits are made starting from the end of the string instead of from the front.
s.splitlines()	Returns a list containing the individual lines of a string split using the newline character \n as the delimiter.

Additional String Splitting Examples

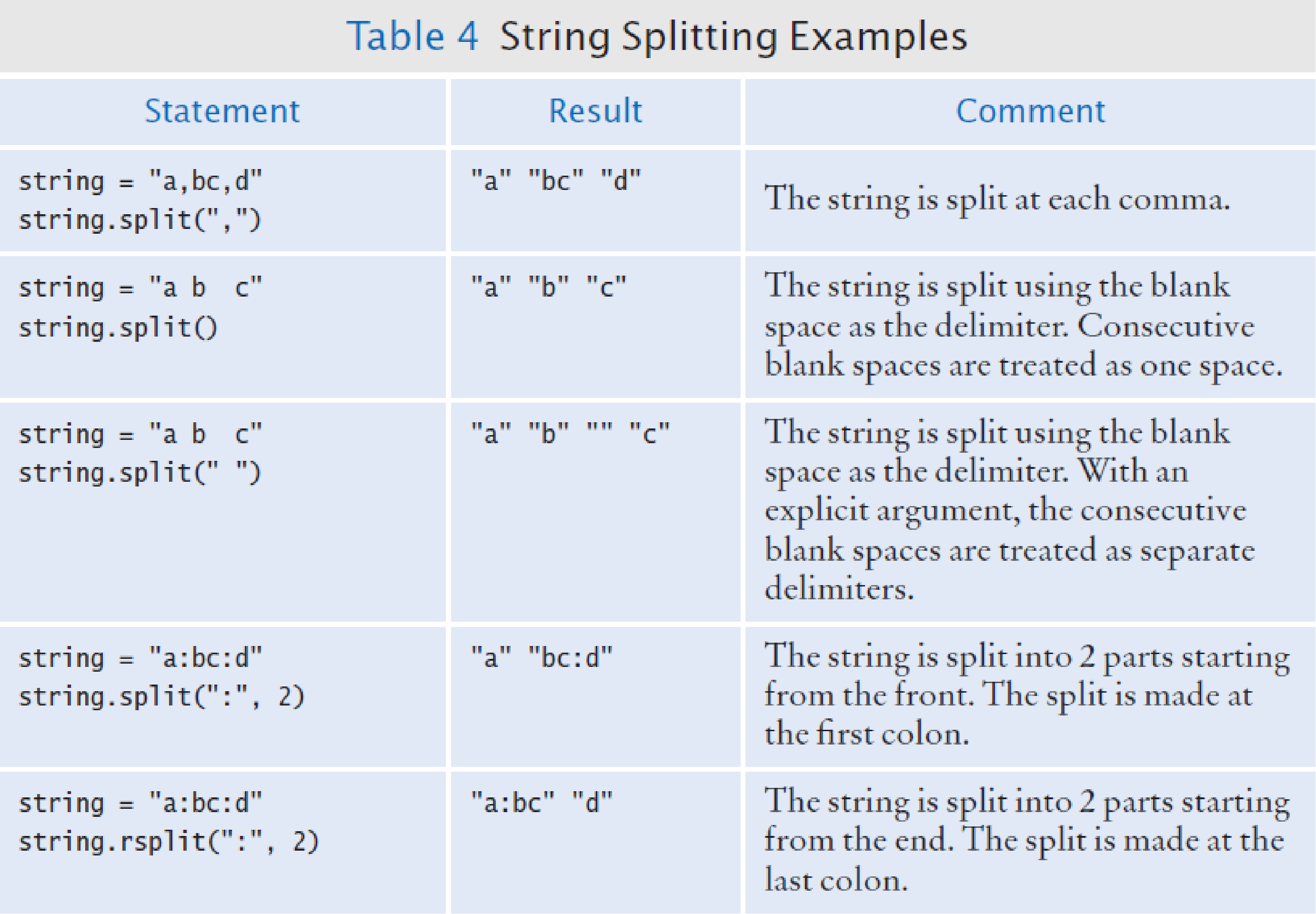


Table 4 String Splitting Examples

Statement	Result	Comment

string = "a,bc,d"
string.split(",")
	"a" "bc" "d"	The string is split at each comma.

string = "a b c"
string.split()
	"a" "b" "c"	The string is split using the blank space as the delimiter. Consecutive blank spaces are treated as one space.

string = "a b c"
string.split(" ")
	"a" "b" "" "c"	The string is split using the blank space as the delimiter. With an explicit argument, the consecutive blank spaces are treated as separate delimiters.

string = "a:bc:d"
string.split(":", 2)
	"a" "bc:d"	The string is split into 2 parts starting from the front. The split is made at the first colon.

string = "a:bc:d"
string.rsplit(":", 2)
	"a:bc" "d"	The string is split into 2 parts starting from the end. The split is made at the last colon.

Reading Characters

- The `read()` method may take a single argument that specifies the number of characters to read
- The method returns a string containing the characters
- When supplied with an argument of 1, the `read()` method returns a string consisting of the next character in the file

```
char = inputFile.read(1)
```

- If the end of the file is reached, it returns an empty string ""

Algorithm: Reading Characters

```
char = inputFile.read(1)
while char != "" :
   Process character
   char = inputFile.read(1)
```

Reading Records

- A text file can contain a collection of data records in which each record consists of multiple fields
  - Usually, one record per line
- Example: a file containing student data consists of records composed of an identification number, name, address, and class year
- When working with text files that contain data records, you generally have to read the entire record before you can process it:

```
For each record in the file:
    Read the entire record
    Process the record
```

Record Formats: One record per line

- Another common format stores each data record on a single line
- If the record's fields are separated by a specific delimiter "`:`" (or other character) you can extract the fields by splitting the line with the `split()` method

```
China:1330044605
India:1147995898
United States:303824646
. . .
```

Record Formats: Example (ugly)

- But what if the fields are not separated by a delimiter?

```
China 1330044605
India 1147995898
United States 303824646
. . .
```

- Because some country names have more than one word, we cannot simply use a blank space as the delimiter because multi-word names would be split incorrectly
- Can we use rsplit in this case?

```
inputString = "United States 303824646"
result = inputString.rsplit(" ",1)
print(result)
```

Record Formats: Example (ugly)

- Another for reading records in this format is to read the line, then search for the first digit in the string returned by readline():

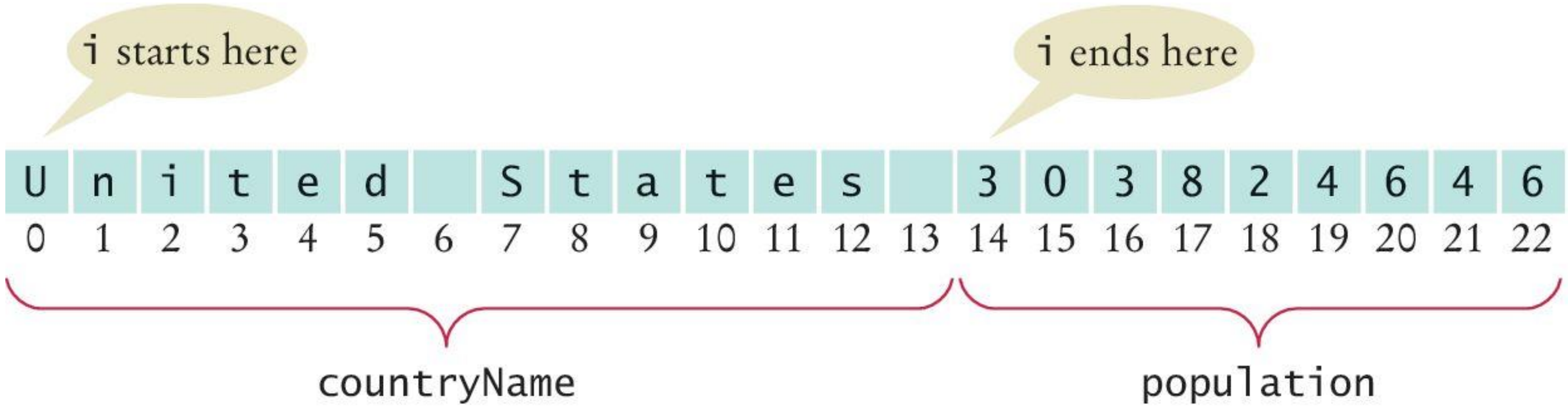
```
i = 0
char = line[0]
while not line[0].isdigit() :
    i = i + 1
```

- You can then extract the country name and population as substrings using the slice operator:

```
countryName = line[0 : i - 1]
population = int(line[i : ])
```

Record Formats: Example (ugly)

- ‘Slicing’ the string read from file



Record Formats: Multiple lines per record

- The organization or format of the records can vary, however, making some formats easier to read than others
- A typical format for such data is to store each field on a separate line of the file with all fields of a single record on consecutive lines:

```
China
1330044605
India
1147995898
United States
303824646
. . .
```

Record Formats: Example

- Reading the data in this format is rather easy
- Because each record consists of two fields, we read two lines from the file for each record

```
line = infile.readline()
while line != "" :
    countryName = line.rstrip()
    line = infile.readline()
    population = int(line)
       Process data record
    line = infile.readline()
```

Reading the whole file – as a “Huge” String

- The `read()` method without any parameter will read the entire file as a single string

```
contents = infile.read()
# the whole file is read in a single (huge!) string
```

Reading the whole file – as a List of Strings

- The `readlines()` method will read the entire file as list of Strings (one string per line)

```
lines = infile.readlines()
```

```
# equivalent to
lines = []
for line in infile:
  lines.append(line)
```

```
# shortcut (not very readable…)
lines = list(infile)
```

File Operations

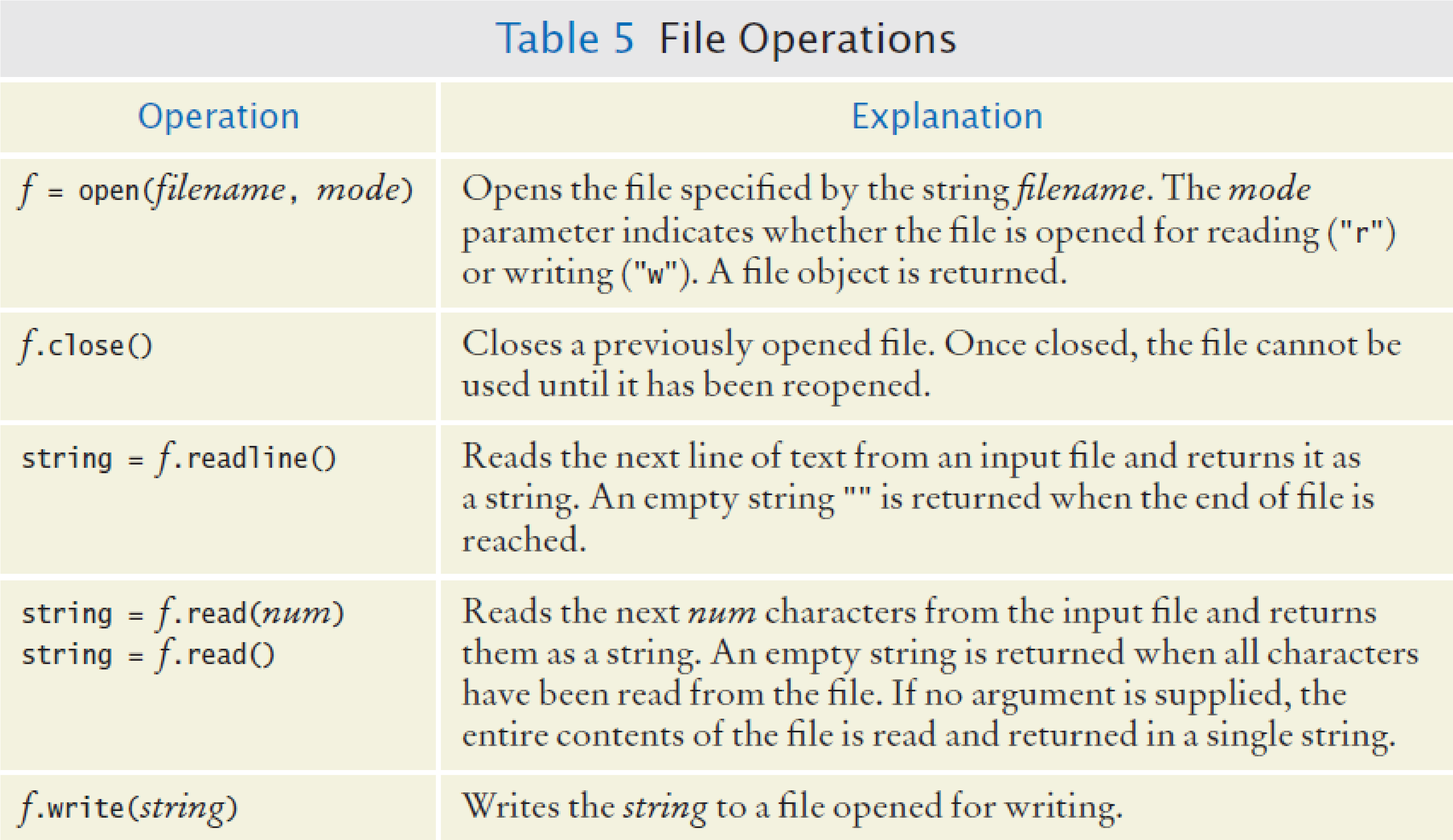


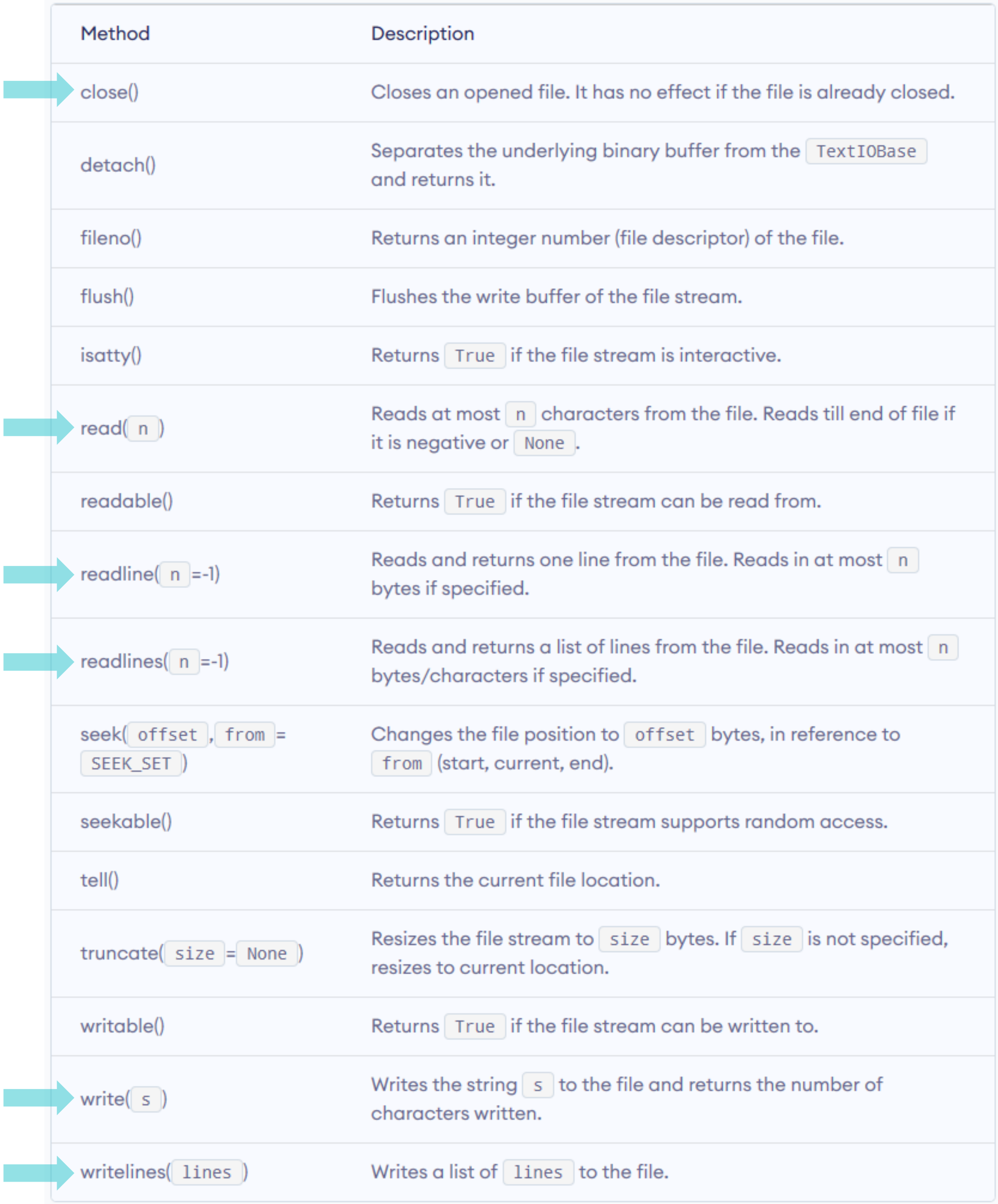
Table 5 File Operations

Operation	Explanation
f `= open(`*filename*`,` *mode*`)`	Opens the file specified by the string *filename*. The *mode* parameter indicates whether the file is opened for reading (`"r"`) or writing (`"w"`). A file object is returned.
f`.close()`	Closes a previously opened file. Once closed, the file cannot be used until it has been reopened.
`string =` *f*`.readline()`	Reads the next line of text from an input file and returns it as a string. An empty string `""` is returned when the end of file is reached.

`string =` *f*`.read(`*num*`)`
`string =` *f*`.read()`
	Reads the next *num* characters from the input file and returns them as a string. An empty string is returned when all characters have been read from the file. If no argument is supplied, the entire contents of the file is read and returned in a single string.

f`.write(`*string*`)`	Writes the *string* to a file opened for writing.

File Operations



Method	Description
close()	Closes an opened file. It has no effect if the file is already closed.
detach()	Separates the underlying binary buffer from the `TextIOBase` and returns it.
fileno()	Returns an integer number (file descriptor) of the file.
flush()	Flushes the write buffer of the file stream.
isatty()	Returns `True` if the file stream is interactive.
read(`n`)	Reads at most `n` characters from the file. Reads till end of file if it is negative or `None`.
readable()	Returns `True` if the file stream can be read from.
readline(`n`=-1)	Reads and returns one line from the file. Reads in at most `n` bytes if specified.
readlines(`n`=-1)	Reads and returns a list of lines from the file. Reads in at most `n` bytes/characters if specified.
seek(`offset`,`from`=`SEEK_SET`)	Changes the file position to `offset` bytes, in reference to `from` (start, current, end).
seekable()	Returns `True` if the file stream supports random access.
tell()	Returns the current file location.
truncate(`size`=`None`)	Resizes the file stream to `size` bytes. If `size` is not specified, resizes to current location.
writable()	Returns `True` if the file stream can be written to.
write(`s`)	Writes the string `s` to the file and returns the number of characters written.
writelines(`lines`)	Writes a list of `lines` to the file.

https://www.programiz.com/python-programming/file-operation

A Second Example

- Open the file items.py

Processing Text Files Example

PAGE 319 – PAGINA 440

Steps to Processing Text Files

- Problem Statement:
  - Read two country data files, `worldpop.txt` and `worldarea.txt`
  - Write a file `world_pop_density.txt` that contains country names and population densities with the country names aligned left and the numbers aligned right

```
Afghanistan          50.56
Akrotiri            127.64
Albania             125.91
Algria               14.18
American Samoa      288.92

.  .  .
```

Six Steps to Processing Text Files

- Understand the Processing Task
  - Process the data "on the go" or store data and then process?
- Determine which files you need to read and write
- Choose a mechanism for obtaining the file names
- Choose between iterating over the file (`for..in file`) or reading individual lines (`while file.readline()`)
  - If all data is on one line, normally use line input
- With line-oriented input, extract required data
  - Examine the line and plan for whitespace, delimiters…
- Use functions to factor out common tasks

Step 1: Understand the Task

- While there are more lines to be read
  - Read a line from each file
  - Extract the country name
  - population = number following the country name in the line from the first file
  - area = number following the country name in the line from the second file
  - If area != 0
    - density = population / area
  - Print country name and density

```
Afghanistan          50.56
Akrotiri            127.64
Albania             125.91
Algria               14.18
American Samoa      288.92

. . .
```

Step 2: Determine the files

- Determine the file you need to read and write
- There are two input files:
  - worldpop.txt
  - worldarea.txt
- There is one output file:
  - world_pop_density.txt

Step 3: Obtaining file names

- Choose a mechanism for obtaining the file names
- We have three options:
  - Hardcode the filename as a constant
  - Prompt the user
  - Use a command line argument (advanced topic)
- We will use hard-coded files names
  - The file names are constants, in this example

Step 4: Iterate or read lines?

- Choose between iterating over the file or reading individual lines
- Generally if the data is grouped in lines, we iterate over the file
- If the data is spread over several lines, we read the individual lines
- In this example we will read individual lines since we are reading from two input files

Step 5: Extract data from lines

- Extract the data into the individual fields
  - Use spilt, rsplit, string slices, etc., to extract the data elements

Step 6: Break up in functions

- Use functions to factor out common tasks
- Find the repetitive tasks and develop functions to handle them

Example Code

- Open the file population.py

Exception Handling 📘 7.4

Exception Handling

- There are two aspects to dealing with run-time program errors:
  - Detecting Errors
  - Handling Errors
- The open function can detect an attempt to read from a non-existent file
  - The open function cannot handle the error
  - There are multiple alternatives, the function does not know which is the correct choice
  - The function reports the error to another part of the program to be handled
- **Exception handling** provides a flexible mechanism for passing control from the error to a handler that can deal with it

Exception Handling: Overview

DETECTING ERRORS (RAISE)

- The program should check whether all conditions are valid and allow normal processing
- Otherwise the program raises an exception
- Statement: **`raise`**
- There are different types of exceptions (`ValueError`, `IOError`, …) depending on the cause
- The exception may also contain a message describing the problem

HANDLING ERRORS (TRY…EXCEPT)

- If you know that some code may generate exceptions, you must define some handler code
- The “controlled” code is put inside a **`try`** block
- The “handler” code is put inside an **`except`** block
- Failure to handle an exception will stop the program

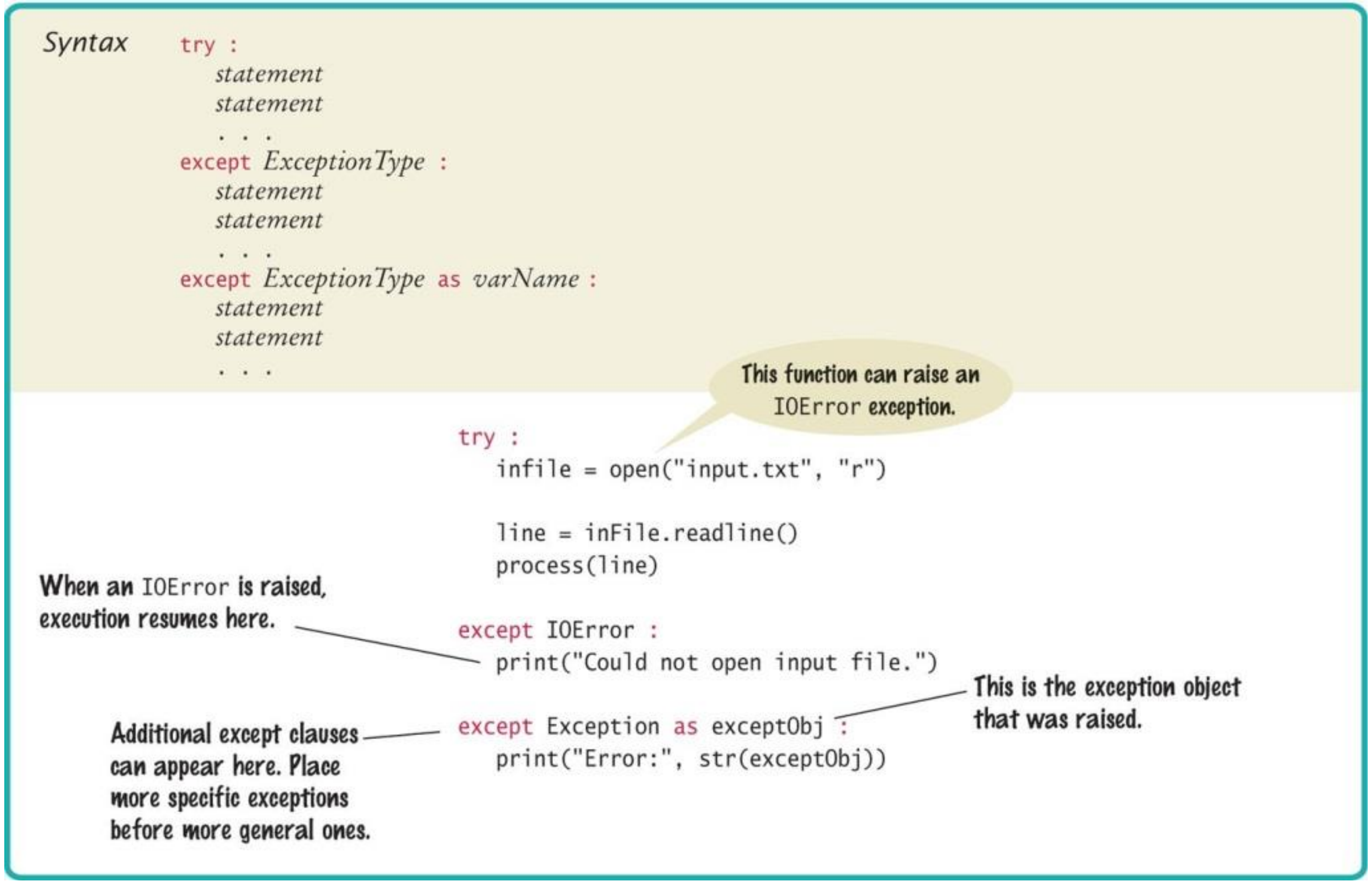
Handling Exceptions

- Every exception should be handled somewhere in the program
- This is a complex problem
  - You need to handle each possible exception and react to it appropriately
  - Not all errors are recoverable
- Handling recoverable errors can be:
  - Simple: exit the program
  - User-friendly: Ask the user to correct the error

Handling Exceptions: Try-Except

- You handle exceptions with the `try`/`except` statement
- Place the statement into a location of your program that knows how to handle a particular exception
- The `try` block contains one or more statements that may cause an exception of the kind that you are willing to handle
- Each `except` clause contains the handler for an exception type

Syntax: Try-Except



Syntax

```
try :
    statement
    statement
    . . .
except ExceptionType :
    statement
    statement
    . . .
except ExceptionType as varName :
    statement
    statement
    . . .
```

This function can raise an IOError exception.

```
try :
   infile = open("input.txt", "r")

   line = inFile.readline()
   process(line)

except IOError :
   print("Could not open input file.")

except Exception as exceptObj :
   print("Error:", str(exceptObj))
```

When an IOError is raised, execution resumes here.

This is the exception object that was raised.

Additional except clauses can appear here. Place more specific exceptions before more general ones.

Try-Except: An Example

```
try :
    filename = input("Enter filename: ")
    infile = open(filename, "r")
    line = infile.readline()
    value = int(line)
. . .
except IOError :
    print("Error: file not found.")
except ValueError as exception :
    print("Error:", str(exception))
```

open() can raise an IOError exception

int() can raise a ValueError exception

Execution transfers here if file cannot be opened

Execution transfers here if the string cannot be converted to an int

If either of these exceptions is raised, the rest of the instructions in the try block are skipped

Example

- If an `IOError` exception is raised, the `except` clause for the `IOError` exception is executed
- If a `ValueError` exception occurs, then second `except` clause is executed
- If any other exception is raised it will not be handled by any of the `except` blocks

Output Messages

- When the body of this handler is executed, it prints the message included with the exception

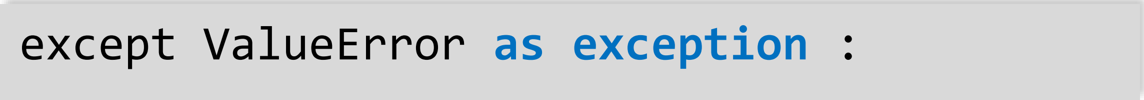
```
except ValueError as exception :
    print("Error:", str(exception))
```

- For example, if the string passed to the `int()` function was `"35x2"`, then the message included with the exception would be:

```
invalid literal for int() with base 10: '35x2'
```

Output Messages (2)

- To obtain the message, we must have access to the exception object itself
- You can store the exception object in a variable with the as syntax:



```
except ValueError as exception :
```

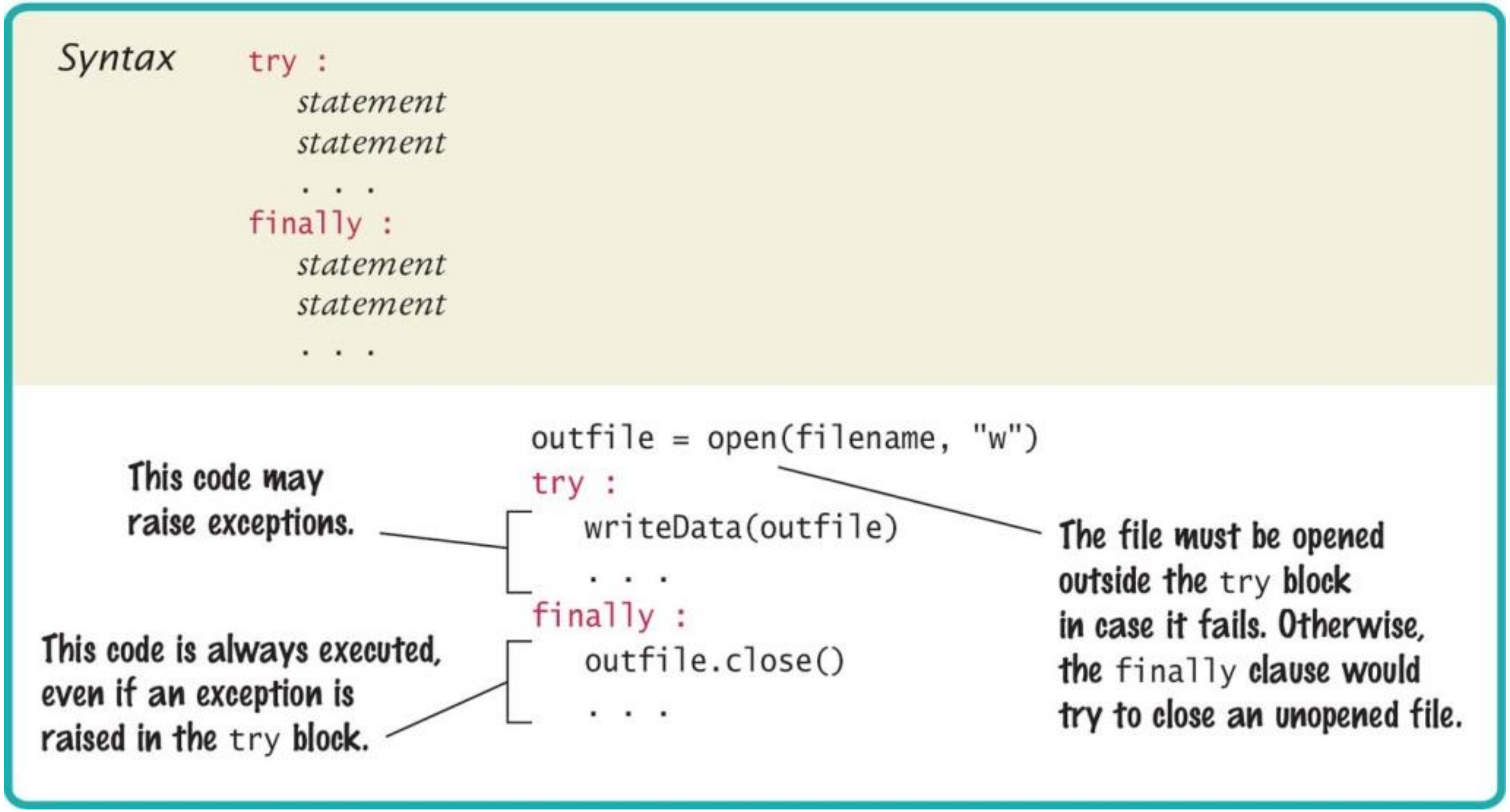
- When the handler for ValueError is executed, `exception` is set to the exception object. In our code, we then obtain the message string by calling `str(exception)`

The finally Clause

- The `finally` clause is used when you need to take some action whether or not an exception is raised
- Here is a typical situation
  - It is important to always close an output file whether or not an exception was raised (to ensure that all output is written to the file)
  - Place the call to `close()` inside a `finally` clause:

```
outfile = open(filename, "w")
try :
    writeData(outfile)
finally :
    outfile.close()
```

Syntax: The Finally Clause



Syntax

```
try :
    statement
    statement
    . . .
finally :
    statement
    statement
    . . .
```

```
outfile = open(filename, "w")
try :
    writeData(outfile)
    . . .
finally :
    outfile.close()
    . . .
```

This code may raise exceptions.

This code is always executed, even if an exception is raised in the try block.

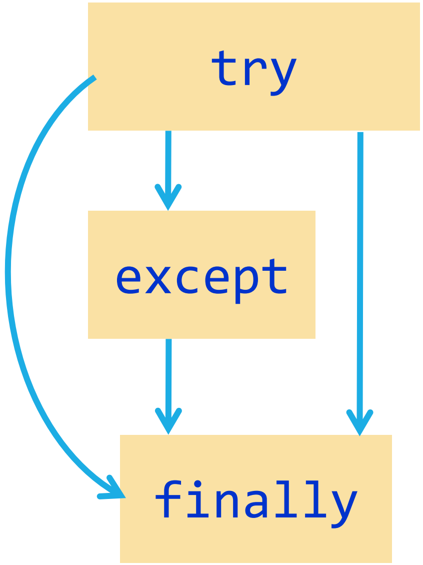
The file must be opened outside the try block in case it fails. Otherwise, the finally clause would try to close an unopened file.

Programming Tip

- Throw exceptions early
  - When a method detects a problem that it cannot solve, it is better to throw an exception rather than try to come up with an imperfect fix
- Catch exceptions late
  - Conversely, a method should only catch an exception if it can really remedy the situation
  - Otherwise, the best remedy is simply to have the exception propagate to its caller, allowing it to be caught by a competent handler

Programming Tip

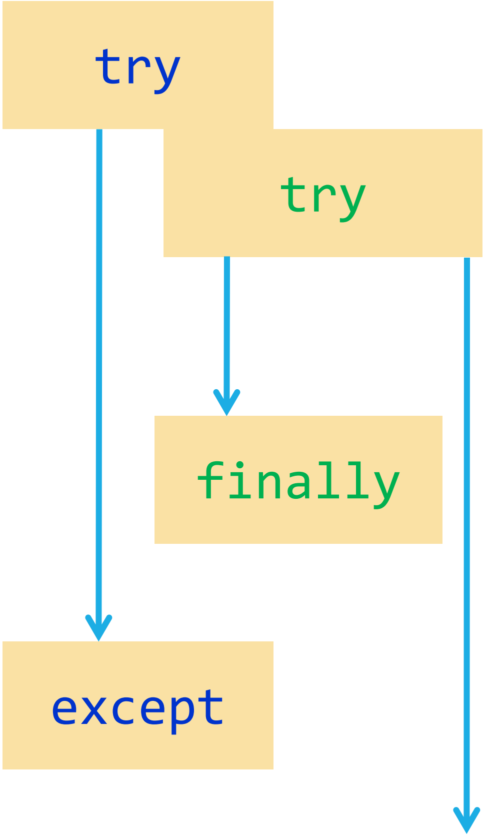
- Do not use except and finally in the same try block
  - The finally clause is executed whenever the try block is exited in any of three ways:
    - 1. After completing the last statement of the try block
    - 2. After completing the last statement of a except clause, if this try block caught an exception
    - 3. When an exception was raised in the try block and not handled



Programming Tip

- It is better to use two (nested) try clauses to control the flow

```
try :
    outfile = open(filename, "w")
  try :
    # Write output to outfile
  finally :
    out.close()  # Close resources
except IOError :
  # Handle exception
```



The With Statement

- Because a `try/finally` statement for opening and closing files is so common, Python has a special shortcut:

```
with open(filename, "w") as outfile :
    Write output to outfile
```

- This `with` statement opens the file with the given name, sets outfile to the file object, *and closes the file object* when the end of the statement has been reached or an exception is raised

Detecting Errors

- What do you do if someone tries to withdraw too much money from a bank account?
- You can “raise” an exception
- When you raise an exception, execution does not continue with the next statement
  - It transfers to the exception handler

Use the `raise` statement to signal an exception

```
if amount > balance :
    raise ValueError("Amount exceeds balance")
```

Source of Output Messages

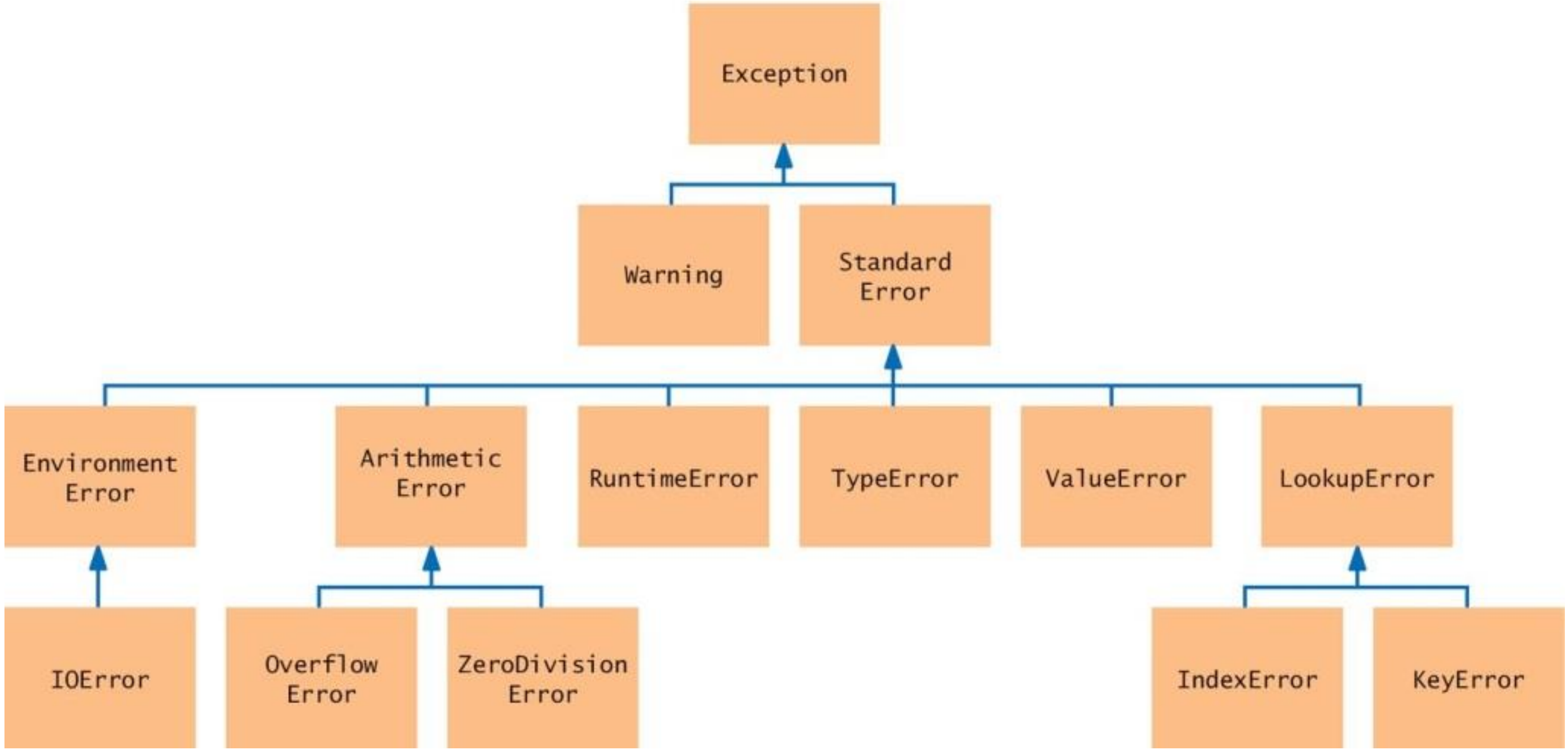
- When you raise an exception, you can provide your own message string. For example, when you call

```
raise ValueError("Amount exceeds balance")
```

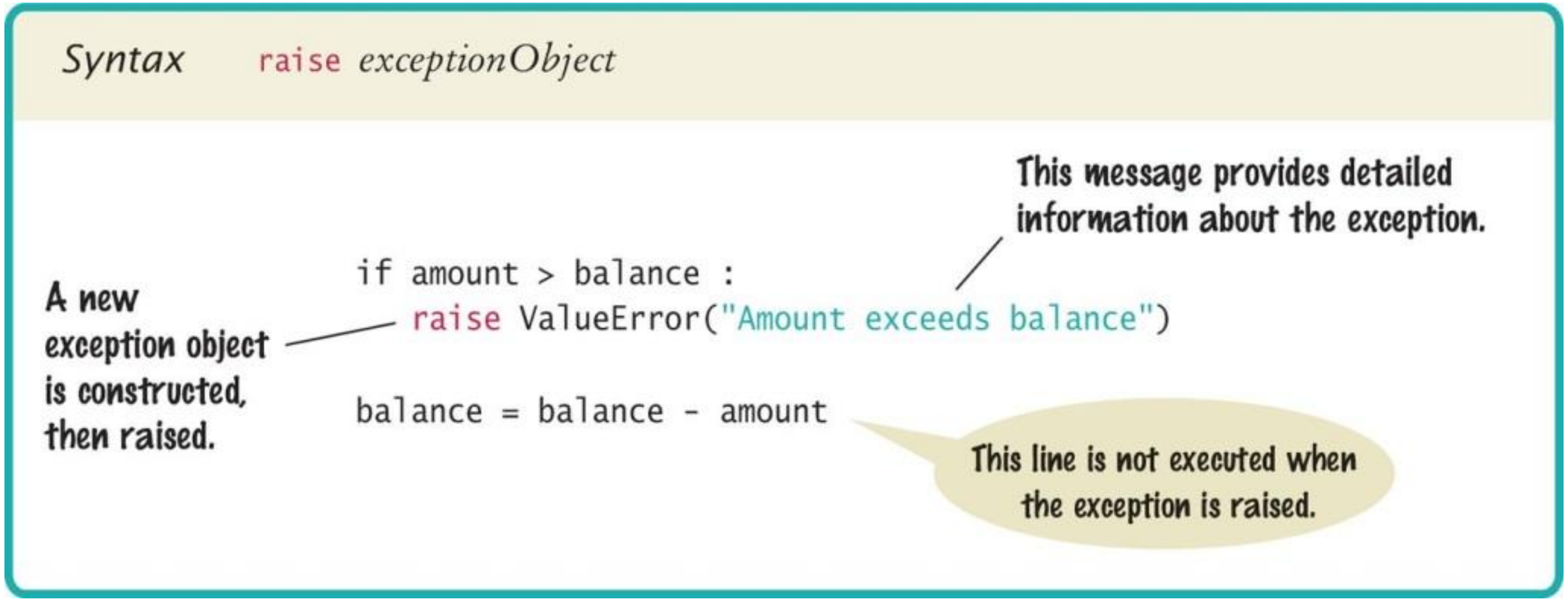
- The message of the exception, `"Amount exceeds balance"`, is the string provided as the argument to the constructor

Exception Classes (a subset)

- Look for an appropriate exception



Syntax: Raising an Exception



Syntax raise *exceptionObject*

```
if amount > balance :
   raise ValueError("Amount exceeds balance")

balance = balance - amount
```

A new exception object is constructed, then raised.

This message provides detailed information about the exception.

This line is not executed when the exception is raised.

Handling Input Errors

📘 7.6

Handling Input Errors

- File Reading Application Example
- Goal: Read a file of data values
  - First line is the count of values
  - Remaining lines have values

```
3
1.45
-2.1
0.05
```

- Risks:
  - The file may not exist
    - The open() function will raise an exception when the file does not exist
  - The file might have data in the wrong format
    - When there are fewer data items than expected, or when the file doesn’t start with the count of values, the program will raise a ValueError exception
    - Finally, when there are more inputs than expected, a RuntimeError exception should be raised

Handling Input Errors: main()

- Outline for method with all exception handling

```
done = False
while not done :
try:
    # Prompt user for file name
    data = readFile(filename)  # May raise exceptions
    # Process data
    done = true;
  except IOError:
      print("File not found.")
  except ValueError :
      print("File contents invalid.")
  except RuntimeError as error:
      print("Error:", str(error))
```

Handling Input Errors: readFile()

- Creates the file object and calls the readData() function
- No exception handling (no except clauses)
- `finally` clause closes file in all cases (exception or not)

```
def readFile(filename) :
    inFile = open(filename, "r") # May throw exceptions
    try
      return readData(inFile)
    finally
      in.close()
```

Handling Input Errors: readData()

- No exception handling (no try or except clauses)
- `raise` creates a `ValueError` exception and exits
- `RuntimeError` exception can occur

```
def readData(inFile) :
    line = inFile.readline()
    numberOfValues = int(line) # May raise a ValueError exception.
    data = []
    for i in range(numberOfValues) :
        line = inFile.readline()
        value = int(line) # May raise a ValueError exception.
        data.append(value)
    # Make sure there are no more values in the file.
    line = inFile.readline()
    # Extra data in file
    if line != "" :
        raise RuntimeError("End of file expected.")
    return data
```

One Possible Scenario

- main calls readFile
  - readFile calls readData
    - readData calls int
    - There is no integer in the input, and int raises a ValueError exception
    - readData has no except clause; it terminates immediately
  - readFile has no except clause; it terminates immediately after executing the finally clause and closing the file
- The IOError except clause is skipped
- The ValueError except clause is executed

Example Code

- Open the file analyzedata.py

Summary

Summary: File Input/Output

- When opening a file, you supply the name of the file stored on disk and the mode in which the file is to be opened
- Close all files when you are done processing them
- Use the `readline()` method to obtain lines of text from a file
- Write to a file using the `write()` method or the `print()` function

Summary: Processing Text Files

- You can iterate over a file object to read the lines of text in the file
- Use the `rstrip()` method to remove the newline character from a line of text
- Use the `split()` method to split a string into individual words
- Read one or more characters with the `read()` method

Summary: Exceptions (1)

- To signal an exceptional condition, use the raise statement to raise an exception object
- When you raise an exception, processing continues in an exception handler
- Place the statements that can cause an exception inside a try block, and the handler inside an except clause
- Once a try block is entered, the statements in a finally clause are guaranteed to be executed, whether or not an exception is raised

Summary: Exceptions (2)

- Raise an exception as soon as a problem is detected
  - Handle it only when the problem can be handled
- When designing a program, ask yourself what kinds of exceptions can occur
- For each exception, you need to decide which part of your program can competently handle it