File Input/Output

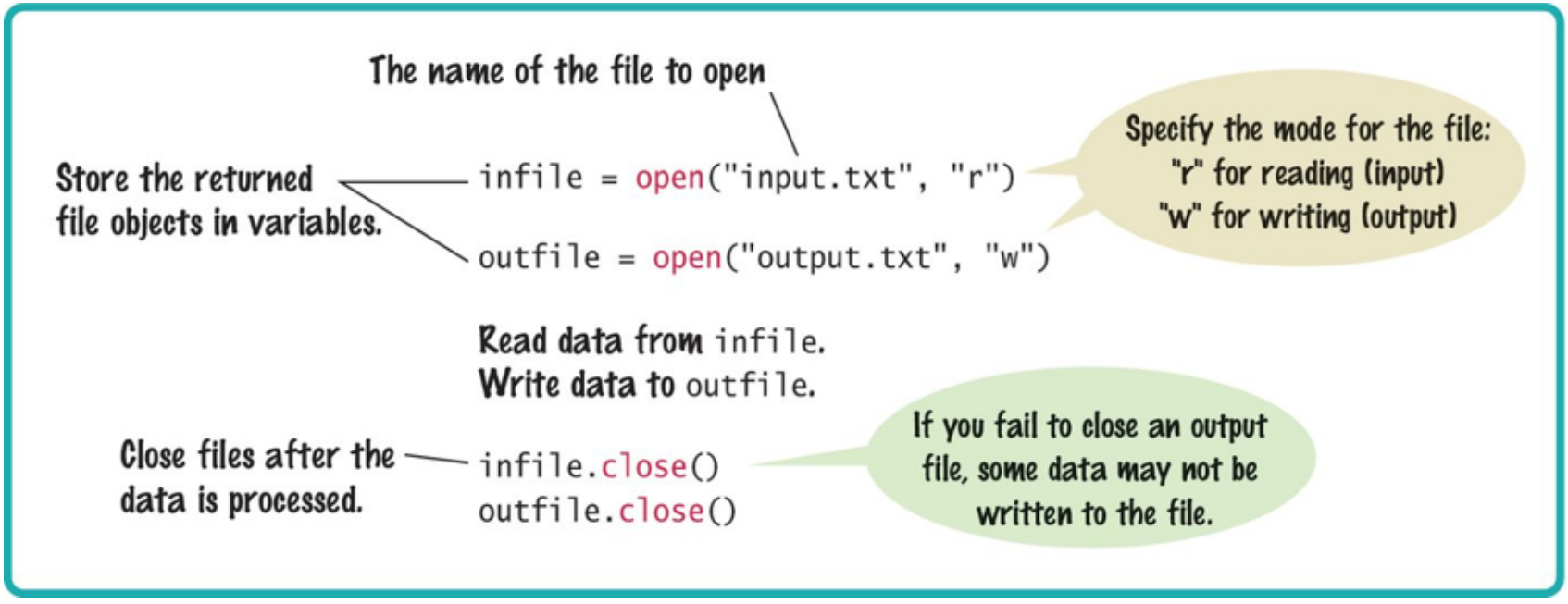
Introduction to Programming

Rae Harbird, r.harbird@ucl.ac.uk

Overview (cont.)

- Python programs connect to files via objects called streams.
- Files themselves are represented by objects of class `File` in Python programs.
- Because they rely on the physical presence and state of actual files and directories, Python programs that use file I/O are prone to unexpected errors
  - . . . and so should include exception handling in their file I/O methods.

Syntax


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.

Other methods

A complete list of methods that you can use with files can be found here.

Method name	Description
`file.close()`	indicates that you are done reading/writing the file
`file.flush()`	writes any buffered data to an open output file
`file.read()`	reads and returns the entire file as a string
`file.readable()`	returns `True` if the file can be read
`file.readline()`	reads and returns a single line of the file as a string
`file.readlines()`	reads and returns the entire file as a list of line strings
`file.seek(position)`	sets the file's current input cursor position
`file.tell()`	returns the file's current input cursor position
`file.writable()`	returns `True` if the file can be written
`file.write("text")`	sends text to an output file
`file.writelines(lines)`	sends a list of lines to an output file

Opening a file

- A **stream** allows for the flow of data between a program and some I/O device or file. When we open a file using Python we are creating a file object that will allow us to use the file.
- An **input stream** allows data flow into the program. For example:

```
inFile = open("input.txt", "r")    #  inFile is a file object
```

- An **output stream** allows data flow out of the program. For example:

```
outFile = open("output.txt", "w")  #  outFile is a file object too
```

Gotcha

- If the file doesn't exist, Python will create the file for you. Be careful though, when you open a file for writing, Python immediately opens it and erases all the current contents. If you want to append content to a file then you should use:

```
appendFile = open("output.txt", "a")
```

- You will get an error if:
  - the file doesn't exist.
  - the file does exist but is read-only and you've asked to write to it.
  - you don't have permission to read or write to the file.
- We will see how to handle these errors later.

Closing a file

After you have finished with a file, you must close it before the program ends. If you don't, the data you have written to the file may not all be saved properly.

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

Reading the whole file

One big string:

```
shoppingListFile = open("shopping_list.txt", "r")
contents = shoppingListFile.read()
print(contents)
```

A list of lines:

```
shoppingListFile = open("shopping_list.txt", "r")
contents = shoppingListFile.readlines()
print(contents)
```

Reading line by line

Python makes it very easy to read a file line by line.

```
shoppingListFile = open("shopping_list.txt", "r")

line = shoppingListFile.readline()
while line != "":
    line = shoppingListFile.readline().rstrip()
    print(line)

shoppingListFile.close()
```

Better still:

```
with open("shopping_list.txt", "r") as file:
    lineCount = 0
    for line in file:
          print("next line:", line.rstrip())
          lineCount += 1
    print("Line count:", lineCount)
```

Note: you have to strip the '\n' at the end of each line off with `rstrip()`. See what happens if you don't.

Token based processing

Sometimes we need to split lines up and process all the elements in a line one by one. This is known as token based processing.

Data is frequently stored as Comma Separated Values (CSV) in text files. For example: the file Trends_in_UK_butterflies.csv.

```
Species of the wider countryside (24),,,,
,Long-term change (1976-2016),Trend,Short-term change (2011-2016),Trend
brimstone,9,No change,105,No change
(Gonepteryx rhamni),,,,
brown argus,-25,No change,13,No change
(Aricia agestis),,,,
comma,138***,Increased,2,No change
(Polygonia c-album),,,,
common blue,-24,No change,1,No change
(Polyommatus icarus),,,,
```

Each line in the file is a record of readings pertaining to one species of butterfly and the fields in the record are separated by commas.

Splitting the line

Let's say that we wanted to read every line of our file containing butterfly data and that we want to extract the name of the butterfly, the latin name of the butterfly, the short-term change and the population trend. Perhaps we might print a report like this:

```
Brimstone (Gonepteryx rhamni):    105, No change
Brown argus (Aricia agestis):     13, No change
Comma (Polygonia c-album):    2, No change
```

. . . . but we need to write an algorithm first . . .

The algorithm

```
Species of the wider countryside (24),,,,
,Long-term change (1976-2016),Trend,Short-term change (2011-2016),Trend
brimstone,9,No change,105,No change
(Gonepteryx rhamni),,,,
brown argus,-25,No change,13,No change
(Aricia agestis),,,,
comma,138***,Increased,2,No change
(Polygonia c-album),,,,
```

butterflies.py

```
    with open(BUTTERFLY_FILE) as file:
        line_count = 0
        butterflyName = ""
        shortTermChange = ""
        trend = 0

        for line in file:
            if line_count >= HEADER_LINES:
                line = line.rstrip()
                if line_count % 2 == 0:
                  butterflyName, long_term_change\
                    long_term_trend, shortTermChange,\
                    trend = line.split(',')
                else:
                    latinName = line.strip(',')
                    print("{} {}:\t {}, {}".format(
                              butterflyName.capitalize(),
                              latinName, shortTermChange.rstrip('*'),
                              trend))
            line_count += 1
        print("\nLine count:", line_count)
```

A problem

Here is a file containing actor's names:

```
Evan Rachel Wood
Will Smith
Oscar Isaac Hernandez Estrada
James Tiberius Kirk
Phillip Seymour Hoffman
Sarah Jessica Parker
Tommy Lee Jones
```

Sometimes lines in a file will not necessarily have the same number of fields.

Likewise, we may only need to process some of the fields in a line and not others.

This won't work:

```
>>> with open("film_stars.txt") as file:
...     first, middle, last = file.read().split()

    Traceback (most recent call last):
      File "<input>", line 2, in <module>
    ValueError: too many values to unpack (expected 3)
```

Using the gather operator '*'

General syntax:

```
name, name, ..., *name = string.split("delimiter")
```

Table 2: Table : Effect of `*extra` at different places

`# extra at start` `*extra, first, last = ...`	`Oscar Isaac Hernandez Estrada` `_________/	|` `		|` ` *extra first last`
`# extra in middle` `first, *extra, last = ...`	`Oscar Isaac Hernandez Estrada` `	_____________/ |` `		|` `first *extra last`
`# extra at end` `first, last, *extra = ...`	`Oscar Isaac Hernandez Estrada` `		_____________/` `		|` `first last *extra`

Note: from Building Python Programs, Obourn, Reges, Stepp

Reading numbers

You may want to process a file, or parts of it, as numbers. How can we do that?

```
# compute.py
# This program adds numbers from a file and prints the sum.

def main():
    sum = 0.0
    with open("bad_numbers.txt") as file:
        for n in file.read().split():
            # try to convert n to a float, but if it
            # is not a valid float, print error message
            try:
                sum += float(n)
            except ValueError:
                print("Invalid number:", n)
    print("Sum is:", round(sum, 1))
```

Writing to a file

We can write the string "Hello, World!" to our output file using the statement:

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

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(f"Number of entries: {count:d}\nTotal: {total:8.2f}\n")
```

Another way of writing to a file

You can also use the `print()` function, as long as you specify where you want the output to go:

```
print("Hello, World!", file=outfile)
```

If you don't want a newline, use the end argument:

```
print("Total: ", end="", file=outfile)
```