Lecture 2.2 : Text files and exceptions

Introduction ¶

- Our Python programs need to be able to save their data to the hard disk. This is *persistent* storage i.e. data saved to the hard disk survives a reboot (unlike data stored in RAM). Our Python programs also need to be able to retrieve data from files on the hard disk. Our programs will, for now, save their data in *text files*. (While text files are human-readable, *binary files* are not. We will cover them later.)
- File processing entails the following steps:
  1. **Open the file:** This step initialises a *file object* that acts as a link from the program to the file on the disk. All subsequent file operations are invoked on the file object. (A file object is sometimes referred to as a *file descriptor* or *stream*.)
  2. **Read and/or write the file:** This is where the work is done. Through the file object the on-disk contents of the file will be read and/or written.
  3. **Close the file:** This step finalises the file and unlinks the file object from the program.

Opening and closing a file

- Below we `open()` a file called `results.txt`. The `r` is the *mode* in which the file has been opened. Here `r` indicates the file has been opened only for reading (it cannot be written to or modified in any way). If the `open()` succeeds (why might it fail?) it returns a reference to a file object which we assign to the variable `f`. When finished with a file we `close()` it.

```
f = open('results.txt', 'r')
f.close()
```

- Once a file has been opened, all subsequent file operations (reading, writing, closing) are carried out by invoking methods on the file object. In the above example the file object is `f`. Here we opened the file for reading. Other modes in which a file can be opened include `w` for writing (warning: if the file already exists when opened for writing it will be truncated i.e. its contents deleted) and `a` for appending (additions to the file will follow any existing contents).

- When we specify a file name in the call to `open()` the Python interpreter will look in the same directory as the program to find the file. If we wish to reference a file that is not in the same directory we need to supply a path to the file e.g. `f = open(r'/tmp/results.txt', 'r')`. The `r` indicates this is a *raw string* and prevents characters such as `/` from taking on any special meaning the Python interpreter might ordinarily assign them.

Reading a file

- There are several methods available to a Python programmer for accessing the contents of a file. The most basic is `read()`:

```
content = f.read()
print(content)
```

- A variant on `read()` is `readlines()`. While `read()` causes the entire content of the file to be assigned to a single string, `readlines()` produces a list of strings, with each entry in the list being a line from the file:

```
content = f.readlines()
print(content)
```

- A potential drawback to `read()` and `readlines()` is that they read in *the entire file contents* and store them in memory. If the file is particularly large this might not be the most efficient use of resources. A sometimes better alternative it to process the file line-by-line. We can read in the contents of a file one line at a time with the `readline()` method (when we reach the end of the file `readline()` sets `line` to the empty string and we exit the loop):

```
line = f.readline()
while line: # The empty string is interpreted as False
    print(line.strip()) # The strip method removes any surrounding whitespace
    line = f.readline()
```

- The simplest way, however, to read a file line-by-line is to use an *iterator*. This approach is similar to using `readline()` but requires less code as an explicit check for the end of the file is not required (the iterator handles that). Here is a program that adopts this approach:

```
f = open('results.txt', 'r')

for line in f:
    print(line.strip())

f.close()
```

File processing

- Each line of `results.txt` consists of a student name and mark. Let's write a program that reads each line from `results.txt` and prints out whether the student in question has passed (or not). Here is an extract from `results.txt`:

```
Joe Murphy 44
Mary Connolly 76
Fred Higgins 30
Laura Timmons 57
```

- We need to read in each line, extract the mark and student name, and print `passed` if the mark is 40+ and `failed` otherwise. The only difficulty is in extracting the name and exam mark from the line. Any ideas on how we can do that? How can we split a line into its constituent strings? How can we merge a list of strings back into a single string? Let's write a function to do the work:

```
import sys

def procfile(filename):

    f = open(filename, 'r')

    for line in f:
        words = line.strip().split() # Note how we can chain methods together
        mark = int(words[-1])
        name = ' '.join(words[0:2])

        if mark >= 40:
            result = 'passed'
        else:
            result = 'failed'

        print('{:s} {:s} with a mark of {:d}'.format(name, result, mark))

    f.close()
```

```
$ python3 procfile_v01.py results.txt
Joe Murphy passed with a mark of 44
Mary Connolly passed with a mark of 76
Fred Higgins failed with a mark of 30
Laura Timmons passed with a mark of 57
```

- This looks OK. But have we tested our program properly? Let’s add two new students to the results file and run the program again:

```
Joe Murphy 44
Mary Connolly 76
Fred Higgins 30
Laura Timmons 57
Fernandinho 22
Mary Lou Ni Bhriain II 87
```

```
$ python3 procfile_v01.py more_results.txt
Joe Murphy passed with a mark of 44
Mary Connolly passed with a mark of 76
Fred Higgins failed with a mark of 30
Laura Timmons passed with a mark of 57
Fernandinho 22 failed with a mark of 22
Mary Lou passed with a mark of 87
```

- Hmm. This is not right. It looks like our program only works correctly when a student’s name consists of two strings. We want to interpret *everything* up to the last entry on each line as the student’s name. We can do so with a simple change to the program:

```
import sys

def procfile(filename):

    f = open(filename, 'r')

    for line in f:
        words = line.strip().split()
        mark = int(words[-1])
        name = ' '.join(words[0:-1])
        if mark >= 40:
            result = 'passed'
        else:
            result = 'failed'

        print('{:s} {:s} with a mark of {:d}'.format(name, result, mark))

    f.close()
```

```
$ python3 procfile_v02.py more_results.txt
Joe Murphy passed with a mark of 44
Mary Connolly passed with a mark of 76
Fred Higgins failed with a mark of 30
Laura Timmons passed with a mark of 57
Fernandinho failed with a mark of 22
Mary Lou Ni Bhriain II passed with a mark of 87
```

Exception handling

- Programs will encounter errors. When something goes wrong we do not want our programs to simply fall over. We want them to be robust to all circumstances that may arise at runtime. How can our programs cope with runtime errors?

- When something goes wrong at runtime the Python interpreter will *raise an exception*. To be robust to runtime errors our code must accept that they will arise from time to time and *handle* resultant exceptions when they are raised. Here are examples of some runtime errors:

```
>>> int('cat')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: invalid literal for int() with base 10: 'cat'
>>> 3/0
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ZeroDivisionError: division by zero
>>> open('nofile.txt')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
FileNotFoundError: [Errno 2] No such file or directory: 'nofile.txt'
```

- The final example above shows the error that arises when we try to open a file that does not exist. Let's look at what happens our function when we pass to it the name of a file that does not exist:

```
$ python3 procfile_v02.py abc.txt
Traceback (most recent call last):
File "procfile_v02.py", line 25, in <module>
main()
File "procfile_v02.py", line 22, in main
procfile(sys.argv[1])
File "procfile_v02.py", line 5, in procfile
f = open(filename, 'r')
FileNotFoundError: [Errno 2] No such file or directory: 'abc.txt'
```

- Rather than have a program abruptly exit on encountering such an error (the default behaviour) Python allows programmers to handle such scenarios with a `try-except` construct.

- In the `try` block of code we place the instructions that may fail due to a runtime error. In the `except` block of code we place the instructions to be carried out in the event of the `try` block failing due to the runtime error. Our new file-opening code that is robust to non-existent files looks like this:

```python
import sys

def procfile(filename):

    try:
        f = open(filename, 'r')

        for line in f:
            words = line.strip().split()
            mark = int(words[-1])
            name = ' '.join(words[0:-1])

            if mark >= 40:
                result = 'passed'
            else:
                result = 'failed'

            print('{:s} {:s} with a mark of {:d}'.format(name, result, mark))

        f.close()

    except FileNotFoundError:
        print('The file {:s} does not exist.'.format(filename))
```

```
$ python3 procfile_v03.py abc.txt
The file abc.txt does not exist.
```

- If an error occurs within the `try` block, execution stops at that point and the rest of the `try` block is ignored. An exception corresponding to the specific error that has arisen is *raised*. Python then searches for an `except` block that can handle the exception.

- If a suitable `except` block is found it is executed and execution continues from the point following the `try-except`. If no suitable `except` block is found then the default behaviour applies and program execution is halted.

- If no error arises in the `try` block then execution continues at the instruction following the `try-except` (the contents of the `except` block are ignored).

- Our code is not yet robust to all runtime errors however. What happens if the file we are processing is incorrectly formatted e.g. it does not contain an integer mark. Then our code will fail again. Assume the file `errors.txt` contains the following entries:

```
Anne O'Brien 55
Paul Quinn 66
Alan Dunne 3e
Joe Lacey 44
Amy Moore 77
```

- Look what happens when we run our code against this file:

```
$ python3 procfile_v03.py errors.txt
Anne O'Brien passed with a mark of 55
Paul Quinn passed with a mark of 66
Traceback (most recent call last):
File "procfile_v03.py", line 29, in <module>
main()
File "procfile_v03.py", line 26, in main
procfile(sys.argv[1])
File "procfile_v03.py", line 10, in procfile
mark = int(words[-1])
ValueError: invalid literal for int() with base 10: '3e'
```

- Hmm. We have a couple of problems here. When the `ValueError` exception is raised our program immediately exits. And it does so *without closing the file*. That's bad practice. Can we fix it so that the file is *always* closed i.e. it is closed when the program runs correctly *and* it is closed in the event of an (unhandled) exception?

- Yes. The use of the `with` statement means the file is always closed cleanly irrespective of whether an exception is raised or not. Let's modify our program to use such a `with` statement:

```
import sys

def procfile(filename):

    try:
        with open(filename, 'r') as f:

            for line in f:
                words = line.strip().split()
                mark = int(words[-1])
                name = ' '.join(words[0:-1])

                if mark >= 40:
                    result = 'passed'
                else:
                    result = 'failed'

                print('{:s} {:s} with a mark of {:d}'.format(name, result, mark

    except FileNotFoundError:
        print('The file {:s} does not exist.'.format(filename))
```

- To handle the `ValueError` exception gracefully however we need a new `except` block. Where should we place it? Well that depends on the kind of behaviour we want. If an error occurs do we want to continue processing the file or do we want to give up at that point? If we assume that we want to continue processing the file at the line following the error then we would do something like this:

```
import sys

def procfile(filename):

    try:
        with open(filename, 'r') as f:

            for line in f:

                try:
                    words = line.strip().split()
                    mark = int(words[-1])
                    name = ' '.join(words[0:-1])

                    if mark >= 40:
                        result = 'passed'
                    else:
                        result = 'failed'

                    print('{:s} {:s} with a mark of {:d}'.format(name, result,
                except ValueError:
                    print('Illegal mark encountered: {}'.format(words[-1]))

    except FileNotFoundError:
        print('The file {:s} does not exist.'.format(filename))
```

```
$ python3 procfile_v05.py errors.txt
Anne O'Brien passed with a mark of 55
Paul Quinn passed with a mark of 66
Illegal mark encountered: 3e
Joe Lacey passed with a mark of 44
Amy Moore passed with a mark of 77
```

- Where would you place the `except` block if you wanted to abandon file processing as soon as a malformed entry in the input file was encountered?

The `else` block

- If execution leaves the `try` block *normally* i.e. **not** as a result of an exception and not through a call to `break` or `return` then the `else` block (if present) is executed. The `else` block must be placed after all `except` blocks.

```
import sys

def procfile(filename):
    try:
        with open(filename, 'r') as f:
            for line in f:
                words = line.strip().split()
                mark = int(words[-1])
                name = ' '.join(words[0:-1])

                if mark >= 40:
                    result = 'passed'
                else:
                    result = 'failed'

                print('{:s} {:s} with a mark of {:d}'.format(name, result, mark

    except ValueError:
        print('Illegal mark encountered: {}'.format(words[-1]))

    except FileNotFoundError:
        print('The file {:s} does not exist.'.format(filename))

    else:
        print('Exited try block normally...')
```

```
$ python3 procfile_v08.py results.txt
Joe Murphy passed with a mark of 44
Mary Connolly passed with a mark of 76
```

```
Fred Higgins failed with a mark of 30
Laura Timmons passed with a mark of 57
Exited try block normally...

$ python3 procfile_v08.py errors.txt
Anne O'Brien passed with a mark of 55
Paul Quinn passed with a mark of 66
Illegal mark encountered: 3e
```

The `finally` block

- A `finally` block is often used in conjunction with `try` and `except` blocks. In the `finally` block we place code that we *always* want executed, irrespective of whether an exception occurs or not. Below we augment our program with a `finally` block that always causes a summary of all successfully processing to be printed before exiting:

```python
import sys

def procfile(filename):
    lines = 0
    try:
        with open(filename, 'r') as f:
            for line in f:

                try:
                    words = line.strip().split()
                    mark = int(words[-1])
                    name = ' '.join(words[0:-1])

                    if mark >= 40:
                        result = 'passed'
                    else:
                        result = 'failed'

                    print('{:s} {:s} with a mark of {:d}'.format(name, result,
                    lines += 1

                except ValueError:
                    print('Illegal mark encountered: {}'.format(words[-1]))

    except FileNotFoundError:
        print('The file {:s} does not exist.'.format(filename))

    finally:
        print('File processing complete')
        print('Lines processed: {}'.format(lines))
```

```
$ python3 procfile_v06.py errors.txt
Anne O'Brien passed with a mark of 55
Paul Quinn passed with a mark of 66
Illegal mark encountered: 3e
Joe Lacey passed with a mark of 44
Amy Moore passed with a mark of 77
File processing complete
Lines processed: 4

$ python3 procfile_v06.py abc.txt
The file abc.txt does not exist.
File processing complete
Lines processed: 0
```

Writing a file

- Let’s modify our program such that it writes its output to a file rather than to the screen. Note how we have enhanced the `with` statement to deal with two files:

```
import sys

def procfile(filename_in, filename_out):
    lines = 0
    try:
        with open(filename_in, 'r') as fin, open(filename_out, 'w') as fout:

            for line in fin:

                try:
                    words = line.strip().split()
                    mark = int(words[-1])
                    name = ' '.join(words[0:-1])

                    if mark >= 40:
                        result = 'passed'
                    else:
                        result = 'failed'

                    fout.write('{:s} {:s} with a mark of {:d}\n'.format(name, r
                    lines += 1

                except ValueError:
                    print('Illegal mark encountered: {}'.format(words[-1]))


    except FileNotFoundError:
        print('The file {:s} does not exist.'.format(filename_in))

    finally:
        print('File processing complete')
        print('Lines processed: {}'.format(lines))
```

```
$ python3 procfile_v07.py errors.txt passes.txt
Illegal mark encountered: 3e
File processing complete
Lines processed: 4

$ cat passes.txt
Anne O'Brien passed with a mark of 55
Paul Quinn passed with a mark of 66
Joe Lacey passed with a mark of 44
Amy Moore passed with a mark of 77
```