ECS 32A - File I/O and Exceptions

Aaron Kaloti

UC Davis - Summer Session #1 2020

Overview

- Reading to files.
- Writing from files.
- Exceptions.

Why Files?

Motivating Example

```
deposit = int(input("How much do you wish to deposit: "))
account_total += deposit
```

- Program suffers from **transience**.
    - Program must turn off eventually -> `account_total`'s value lost.

Other Uses

- Storing data.
- Transferring data.

Reading from a File

Example: Reading Account Balance

Code

```python
# Read balance from file.
file = open("balance.txt")
account_total = int(file.read())
print("Current balance: {}".format(account_total))

# Perform the user's deposit.
deposit = int(input("How much do you wish to deposit: "))
account_total += deposit
print("Current balance: {}".format(account_total))

file.close()
```

bank-deposit-read-balance.py

Text File

```
80
```

balance.txt

Executions

```
Current balance: 80
How much do you wish to deposit: 7
Current balance: 87
```

```
Current balance: 80
How much do you wish to deposit: 15
Current balance: 95
```

Remarks

- Opened file with `open()` (next slide).
- `balance.txt` is assumed to be in same folder as this Python file.
- Reading values from file reads them in as strings.

Opening a File

Description of `open()`

- Takes two arguments.
  1. File name (more accurately called the "file path").
  2. (optional) The mode.
- Returns "file object" / "file handle".
- Four different modes (default is `"r"`):
  1. `"r"` - Read - Default value. Opens a file for reading, error if the file does not exist.
  2. `"a"` - Append - Opens a file for appending.
  3. `"w"` - Write - Opens a file for writing.
  4. `"x"` - Create - Creates the specified file. Returns an error if the file exists.
  - Both append and write create the file if it doesn't exist.

Opening a File

Only One Mode Allowed

```
>>> f = open("balance.txt", "rw")
...
ValueError: must have exactly one of create/read/write/append mode
>>> f = open("balance.txt", "aw")
...
ValueError: must have exactly one of create/read/write/append mode
```

Opening a File

If Have Wrong File Path

```
file = open("poem2.txt")
print(file.read())
file.close()
```

```
Traceback (most recent call last):
  File "/home/aaronistheman/.10demos/read-poem.py", line 1, in <module>
    file = open("poem2.txt")
FileNotFoundError: [Errno 2] No such file or directory: 'poem2.txt'
```

Reading from a File

`read()`

- Reads the file's entire contents (into one string).
- Each successive call returns an empty string, since the file has already been read.

Example

```python
file = open("poem.txt")
print("=== File contents ===")
print(file.read())
print("=== File contents ===")
print(file.read())
print("=== File contents ===")
print(file.read())
file.close()
```

read-poem.py

```
There once was a man from Peru;
who dreamed he was eating his shoe.
He woke with a fright
in the middle of the night
to find that his dream had come true.
```

poem.txt

- Output:

```
=== File contents ===
There once was a man from Peru;
who dreamed he was eating his shoe.
He woke with a fright
in the middle of the night
to find that his dream had come true.

=== File contents ===

=== File contents ===
```

Reading from a File

`readline()`

- Requires `while` loop.
- Similar to how file is read in C programming language.

Example

```
file = open("poem.txt")
line = file.readline()
line_num = 1
while line != "":
    print("Line {}: {}".format(line_num, line),end='')
    line = file.readline()
    line_num += 1
file.close()
print("end")
```

read-poem-each-line.py

- Output:

```
Line 1: There once was a man from Peru;
Line 2: who dreamed he was eating his shoe.
Line 3: He woke with a fright
Line 4: in the middle of the night
Line 5: to find that his dream had come true.
end
```

Reading from a File

`readlines()`[1]

```python
file = open("poem.txt")
lines = file.readlines()
print("From readlines(): {}\n".format(lines))
for line in lines:
    line = line[:-1]  # cut off newline character
    print(line)
file.close()
```

read-poem-each-line-readlines.py

```
From readlines(): ['There once was a man from Peru;\n', 'who dreamed he was eating h
is shoe.\n', 'He woke with a fright\n', 'in the middle of the night\n', 'to find tha
t his dream had come true.\n']

There once was a man from Peru;
who dreamed he was eating his shoe.
He woke with a fright
in the middle of the night
to find that his dream had come true.
```

1. This *differs* from the previous slide: `readline()` vs. `readlines()`.

Reading from a File

for Loop Without readlines()

```
file = open("poem.txt")
for line in file:
    print(line,end='')
file.close()
```

read-poem-each-line-for-loop.py

```
There once was a man from Peru;
who dreamed he was eating his shoe.
He woke with a fright
in the middle of the night
to find that his dream had come true.
```

Example: Print Lines Bigger Than 4

Prompt

- Write a function that takes a file name as argument and prints each line in that file that has a length greater than 4.

Solution #1

```
def print_bigger_than_4(filename):
    f = open(filename)
    for line in f:
        line = line[:-1]
        if len(line) > 4:
            print(line, end='')
    f.close()
```

Solution #2

```
def print_bigger_than_4(filename):
    f = open(filename)
    line = f.readline()
    while line != "":
        if len(line) > 5:
            print(line, end='')
        line = f.readline()
    f.close()
```

Writing to a File

`write()`

Example

```python
file = open("testfile.txt","w")
file.write("abc")
file.write("def")
file.close()
```
my-first-write.py

```
blah blah blah blah
```
testfile.txt

- After running the above program, `textfile.txt` will contain:

```
abcdef
```
testfile.txt

Remarks

1. Opening the file for writing *immediately* erases its contents.
2. `write()` doesn't automatically add newline character.

Closing a File

- Should always close a file once done using it.
- Doesn't always/consistently matter.
    - If you don't close a file, your changes to it (if you write to the file) may not be saved.
    - Other reasons relating to operating systems concepts that you should close files.

Example: Bank Account

- We'll modify bank account example to read old balance *and* write new balance.

Reading Old Balance

- Read current balance as before, but close because need to reopen with different mode.

```
# Read balance from file.
readfile = open("balance.txt")
balance = int(readfile.read())
readfile.close()
print("Current balance: {}".format(balance))
```

Example: Bank Account

Ask For Deposit

- As before, ask for deposit and adjust accordingly.

```
# Perform the user's deposit.
deposit = int(input("How much do you wish to deposit: "))
balance += deposit
print("Current balance: {}".format(balance))
```

Example: Bank Account

Writing New Balance

- Open file for writing.

```
writefile = open("balance.txt", "w")
```

- Write the balance.
    - Must use `str()` because `balance` is integer.

```
# Save the balance to the file.
writefile.write(str(balance))
writefile.close()
```

Example: Bank Account

Final Program

```python
# Read balance from file.
readfile = open("balance.txt")
balance = int(readfile.read())
readfile.close()
print("Current balance: {}".format(balance))

# Perform the user's deposit.
deposit = int(input("How much do you wish to deposit: "))
balance += deposit
print("Current balance: {}".format(balance))

# Save the balance to the file.
writefile = open("balance.txt", "w")
writefile.write(str(balance))
writefile.close()
```

bank-deposit-read-write-balance.py

Example: Copy and Sum

Prompt

Write a function that takes two file paths as its arguments. You may assume that the first file (i.e. the file whose path is given by the first argument) will *always only contain integers, with one integer per line*, as shown in the sample below:

```
5
8
1
2
```

`copy_and_sum()` should copy the contents of the first file into the second file (whose path is given by the second argument), completely overwriting the old contents of the second file[1]. Additionally, the function should also return the sum of all integers copied over.

For example, if the sample file above is `foo1.txt`, then `copy_and_sum("foo1.txt","foo2.txt")` should return 16, because that is the sum of 5, 8, 1, and 2. Morever, after running the function, `foo2.txt` should have the exact same contents as `foo1.txt`.

1. Why I point this out will become apparent once we learn about appending to a file.

Example: Copy and Sum

Solution

```
def copy_and_sum(filename1, filename2):
    readfile = open(filename1)
    writefile = open(filename2, "w")
    s = 0
    for line in readfile:
        s += int(line)
        writefile.write(line)
    readfile.close()
    writefile.close()
    return s
```

Note on Methods

- `readline()`, `read()`, `readlines()`, `write()`, and `close()` are methods.
    - Recall: calling a method is done with the notation *`object.method_name(arguments)`*.

Failing to Open a File

Examples of Reasons

1. The file does not exist, or the file path was wrong.
2. Permission errors.
3. Opening a directory (i.e. the more commonly used word for "folder", in computer science).

As Python Code

- From section 14.5 of your book[1]:

```
>>> fin = open('bad_file')
FileNotFoundError: [Errno 2] No such file or directory: 'bad_file'
>>> fout = open('/etc/passwd', 'w')
PermissionError: [Errno 13] Permission denied: '/etc/passwd'
>>> fin = open('/home')
IsADirectoryError: [Errno 21] Is a directory: '/home'
```

- Note: all of these are crashes; failing to open a file *ends the program immediately*.

1. The original slide had `IOError` where the slide shows `FileNotFoundError`, because the textbook that I copied the above example from used `IOError`. As far back as Python 3.3, `IOError` is no longer used, and since the book uses Python 3.4, I suspect that the book has a typo / example that needs to be updated. I have emailed the book's author about this.

Failing to Open a File

Why Crashing Immediately May be Undesirable

- May not ruin the goal of the program.
  - Example: program could use default values instead of what would've been read from the file.
- May want to print more informative error messages (end "gracefully").
- May have to clean up certain resources before ending.

Recovering from a Crash

- Let's take a look at exceptions.

Exceptions

- An **exception** indicates a certain kind of error (with an optional associated message) and forces a program to address the error or crash.

Example #1

```python
a = 8
b = 0
print(a / b)
```

foo.py

```
Traceback (most recent call last):
  File "foo.py", line 3, in <module>
    print(a / b)
ZeroDivisionError: division by zero
```

- The line `print(a / b)` (specifically the operation `a / b`) raised/threw[1] a `ZeroDivisionError` exception.

Example #2

```python
a = b - 4
```

goo.py

```
Traceback (most recent call last):
  File "goo.py", line 1, in <module>
    a = b - 4
NameError: name 'b' is not defined
```

- The line `a = b - 4` raised/threw a `NameError` exception.

1. "raise" is the Python term. In C/C++, the term "throw" is used instead.

Catching Exceptions

- Usually, when an exception is thrown, the program crashes.
- Use `try`/`except` clauses to prevent this.
  - `try` clause: try to execute all of the code in this block. If successful, then ignore the `except` clause.
  - `except` clause: *immediately* go this clause the moment an exception is raised/thrown in the `try` clause.

Example #1

```python
try:
    print("AAA")
    a = 5 / 0
    print("BBB")
except:
    print("You divided by zero, dummy!")
```

my-first-exception.py

- Output:

```
AAA
You divided by zero, dummy!
```

Catching Exceptions

- Avoids a crash, so lines after the `try`/`except` clauses can be executed.

Example #2

```python
try:
    val = int(input("Enter: "))
    print("You entered:", val)
except:
    print("You did not enter an integer.")

print("This will always be printed.")
```

my-second-exception.py

Executions

```
Enter: 5
You entered: 5
This will always be printed.
```

```
Enter:     5
You entered: 5
This will always be printed.
```

```
Enter: 5a
You did not enter an integer.
This will always be printed.
```

Catching Exceptions

- Exceptions not thrown in a `try` block won't be caught.

Example

```python
a = 5 / 0
try:
    print(a)
except:
    print("Dividing by zero is so not cool!")

print("This won't get printed.")
```

raised-too-early.py

Execution

```
Traceback (most recent call last):
  File "raised-too-early.py", line 1, in <module>
    a = 5 / 0
ZeroDivisionError: division by zero
```

Raising Exceptions Explicitly[1]

- Can explicitly raise exception with `raise` keyword.
  - The specific exception that you raise (`ZeroDivisionError`, `NameError`, `ValueError`, etc.) doesn't have to make sense, but you should try to use an appropriate one.
- Useful when writing code for other people to use.
  - This will make more sense once we get to user-defined classes.

Example #1

```python
print("This program will intentionally crash right about now.")
raise ValueError
print("This line won't be reached.")
```

raise0.py

- Output:

```
This program will intentionally crash right about now.
Traceback (most recent call last):
  File "raise0.py", line 2, in <module>
    raise ValueError
ValueError
```

1. These slides on raising exceptions explicitly did not exist during the 07/22 lecture, but I added them in order to clarify what I explained about the example on slide 34.

Raising Exceptions Explicitly

Example #2

```python
try:
    print("AAA")
    raise ZeroDivisionError
    print("BBB")
except:
    print("CCC")
print("DDD")
```

raise1.py

- Output:

```
AAA
CCC
DDD
```

Raising Exceptions Explicitly

- Can attach message to the exception and then raise it.
    - Again, although the message needn't be relevant, you should try to make it relevant.

Example #3

```python
print("This program will intentionally crash right about now.")
raise FileNotFoundError("Fish tacos!")
print("This line won't be reached.")
```

raise2.py

- Output:

```
This program will intentionally crash right about now.
Traceback (most recent call last):
  File "raise2.py", line 2, in <module>
    raise FileNotFoundError("Fish tacos!")
FileNotFoundError: Fish tacos!
```

Propagating[1]

- An exception not caught in a function will *propagate* outside of the function, where it still can be caught. That is, the function that throws the exception does not need to be the one to catch it.

Example #1

```python
def foo(a, b):
    c = a / b
    return c

try:
    print("AAA")
    foo(2, 0)
    print("BBB")
except:
    print("Caught exception.")
```

- Output:

```
AAA
Caught exception.
```

1. These slides also did not exist during the 07/22 lecture but were created to clarify the example on slide 34.

Propagating

- If an exception is raised *and* caught within the same function, it won't propagate outside of the function. The caller will never know or be able to check that an exception was raised/caught in the function it called.

Example #2

```python
def foo(a, b):
    try:
        c = a / b
        return c
    except:
        return -1

try:
    print("AAA")
    foo(2, 0)
    print("BBB")
except:
    print("Caught exception.")
```

```
AAA
BBB
```

Catching Specific Exceptions

Example #1

```python
try:
    a = int(input("Enter: "))
    b = int(input("Enter: "))
    c = a / b
    foo()       # nonexistent function
except ZeroDivisionError:
    print("You divided by zero!")
except ValueError:
    print("You did not enter a valid integer!")
except:
    print("Something truly bizarre happened!")
```

catching-specific-exceptions.py

Executions

```
Enter: 3
Enter: 0
You divided by zero!
```

```
Enter: 3a
You did not enter a valid integer!
```

```
Enter: 3
Enter: 5
Something truly bizarre happened!
```

Catching Specific Exceptions

Example #2

```python
def bar():
    try:
        f = open("blajasjfklasjl")
    except:
        print("EXCEPTION")
        raise ValueError

try:
    a = int(input("Enter: "))
    b = int(input("Enter: "))
    c = a / b
    bar()
except ZeroDivisionError:
    print("You divided by zero!")
except ValueError:
    print("You did not enter a valid integer!")
except:
    print("Something truly bizarre happened!")
```

catching-specific-exceptions2.py

Execution

```
Enter: 3
Enter: 5
EXCEPTION
You did not enter a valid integer!
```

Remarks

- If the `raise ValueError` were removed, then the `except ValueError` exception handler would not have been triggered, because no exception would *propagate* out of `bar()`.

Raising/Throwing Exceptions

- When you `raise` an exception, you can specify the message associated with it. Compare the three scenarios shown below.

```
>>> a = 5 / 0
Traceback (most recent call last):
  File "<pyshell#0>", line 1, in <module>
    a = 5 / 0
ZeroDivisionError: division by zero
>>> raise ZeroDivisionError
Traceback (most recent call last):
  File "<pyshell#1>", line 1, in <module>
    raise ZeroDivisionError
ZeroDivisionError
>>> raise ZeroDivisionError("iowjwiojioajls")
Traceback (most recent call last):
  File "<pyshell#2>", line 1, in <module>
    raise ZeroDivisionError("iowjwiojioajls")
ZeroDivisionError: iowjwiojioajls
```

Failing to Open a File

Recovering with Exceptions

First Observation: Files Throw Exceptions

```
>>> fin = open('bad_file')
FileNotFoundError: [Errno 2] No such file or directory: 'bad_file'
>>> fout = open('/etc/passwd', 'w')
PermissionError: [Errno 13] Permission denied: '/etc/passwd'
>>> fin = open('/home')
IsADirectoryError: [Errno 21] Is a directory: '/home'
```

- Since there are all kinds of different exceptions that can be thrown during an error in opening a file, it is usually best to just catch all exceptions, rather than specific ones.

Failing to Open a File

Recovering with Exceptions

Example

```
try:
    f = open("non_existent_file")
    # ... read from file ...
except:
    print("Failed to open file, but let's keep going anyways.")

print("blah blah blah")
```

Execution

```
Failed to open file, but let's keep going anyways.
blah blah blah
```

Example: Bank Account

Reacting to Nonexistent `balance.txt` File

- Assume default balance of 0 in this case.

```python
# Read balance from file.
try:
    readfile = open("balance.txt")
    balance = int(readfile.read())
    readfile.close()
except:
    balance = 0
print("Current balance: {}".format(balance))

# Perform the user's deposit.
deposit = int(input("How much do you wish to deposit: "))
balance += deposit
print("Current balance: {}".format(balance))

# Save the balance to the file.
writefile = open("balance.txt", "w")
writefile.write(str(balance))
writefile.close()
```

bank-deposit-exceptions.py

Executions

```
Current balance: 0
How much do you wish to deposit: 50
Current balance: 50
```

```
Current balance: 50
How much do you wish to deposit: 80
Current balance: 130
```

Example: Bank Account

Reacting to Nonexistent `balance.txt` File

- Default value needn't be set in `except` clause.

```
# Read balance from file.
balance = 0
try:
    readfile = open("balance.txt")
    balance = int(readfile.read())
    readfile.close()
except:
    pass
print("Current balance: {}".format(balance))

# Perform the user's deposit.
deposit = int(input("How much do you wish to deposit: "))
balance += deposit
print("Current balance: {}".format(balance))

# Save the balance to the file.
writefile = open("balance.txt", "w")
writefile.write(str(balance))
writefile.close()
```

bank-deposit-exceptions2.py

- Still need `except` clause to prevent crash (i.e. to prevent exception from *propagating*).

Example: Bank Account

Reacting to Nonexistent `balance.txt` File

Note on Extensive Error-Checking

```
...

# Save the balance to the file.
writefile = open("balance.txt", "w")
writefile.write(str(balance))
writefile.close()
```

bank-deposit-exceptions2.py

- May be beneficial to include `try`/`except` setup here, because creating a file can fail.
- In certain classes (e.g. ECS 150), rigorous error-checking may be expected. In this class, I will always ask for error-checking *explicitly* (if at all).

Example: Copy and Sum (Revisited)

Prompt

- Modify `copy_and_sum()` so that it doesn't crash if the file doesn't have an integer on certain lines.
    - Example: if the below file were the input, then only the `5` and `17` would be considered.

```
5
3a
abc
17
```

testfile.txt

Solution

```python
def copy_and_sum(filename1, filename2):
    readfile = open(filename1)
    writefile = open(filename2, "w")
    s = 0
    for line in readfile:
        try:
            s += int(line)
            writefile.write(line)
        except:
            pass
    readfile.close()
    writefile.close()
    return s
```

Appending to a File

Motivation

- Recall: opening a file for writing immediately clears contents, e.g. `f = open("file.txt","w")` immediately clears the contents of `file.txt`.
- Sometimes, may prefer to *append* to (i.e. write to the end of) a file instead. To do so, use the append mode, i.e. `f = open("file.txt","a")`. Opening `file.txt` will *not* clear the file immediately, and successive writes to `file.txt` will go to the end[1] of the file.

1. There is also a `seek()` method that can be used to write to the middle of a file or anywhere besides the end.

Example: Bank Account

Logging

- Log each deposit to `log.txt`
    - Example: if user deposits 20 dollars, then the message "Deposit: 20" should be logged.

```python
# Read balance from file.
balance = 0
try:
    readfile = open("balance.txt")
    balance = int(readfile.read())
    readfile.close()
except:
    pass
print("Current balance: {}".format(balance))


# Perform the user's deposit.
deposit = int(input("How much do you wish to deposit: "))
balance += deposit
print("Current balance: {}".format(balance))

# Save the balance to the file.
writefile = open("balance.txt", "w")
writefile.write(str(balance))
writefile.close()

# Log the deposit.
logfile = open("log.txt", "a")
logfile.write("Deposit: {}\n".format(deposit))
logfile.close()
```

bank-deposit-logging.py

Example: Bank Account

Logging

Executions

```
Current balance: 3
How much do you wish to deposit: 5
Current balance: 8
```

```
Current balance: 8
How much do you wish to deposit: 20
Current balance: 28
```

Log File Afterwards

```
Deposit: 5
Deposit: 20
```

log.txt

The `with` Operator

- Can open a file that'll automatically close once exit `with` clause.

Example: Bank Account

```python
# Read balance from file.
balance = 0
try:
    with open("balance.txt") as readfile:
        balance = int(readfile.read())
except:
    pass
print("Current balance: {}".format(balance))

# Perform the user's deposit.
...
```

bank-deposit-with.py

- Even if an exception is thrown during the call to `int()`[1], the `with` operator will *ensure* that `balance.txt` is properly closed.

1. Can occur if `balance.txt` exists but is empty or doesn't contain integer.

The `with` Operator

- `with` operator is separate concept from exceptions.

Example

```python
with open("balance.txt") as readfile:
    print("Closed 1: {}".format(readfile.closed))
    a = 5 / 0
    print("Closed 2: {}".format(readfile.closed))
print("Closed 3: {}".format(readfile.closed))
```

Execution

- File is closed, but `ZeroDivisionError` is uncaught.

```
Closed 1: False
Traceback (most recent call last):
  File "/home/aaronistheman/Documents/education/teaching/32a/lectures/slides/10/demo
s/with.py", line 3, in <module>
    a = 5 / 0
ZeroDivisionError: division by zero
>>> readfile.closed
True
```

The `with` Operator

Example

- Can add `try`/`except` clauses if we want to.

```python
try:
    with open("balance.txt") as readfile:
        print("Closed 1: {}".format(readfile.closed))
        a = 5 / 0
        print("Closed 2: {}".format(readfile.closed))
    print("Closed 3: {}".format(readfile.closed))
except:
    print("Caught!")
```

with2.py

Execution

```
Closed 1: False
Caught!
```

Appendix: `finally` Clause

- Python has a `finally` clause that allows the programmer to specify something that should *always* happen after a corresponding `try` clause finishes, regardless of whether an exception was raised in the `try` clause or not.

```
>>> try:
    a = 5 / 0
finally:
    print("Oh no!")

Oh no!
Traceback (most recent call last):
  File "<pyshell#16>", line 2, in <module>
    a = 5 / 0
ZeroDivisionError: division by zero

>>> try:
    a = 5 / 3
finally:
    print("Oh no!")

Oh no!
```

Appendix: `finally` Clause

- I believe this is used more in languages like Java, for the purpose of closing files in cases of error. (Since Python has the `with` operator, which some consider to be a wrapper around `finally`, I can't think of reasons to use `finally` in Python.)
- Further reading: https://docs.python.org/3/tutorial/errors.html#defining-clean-up-actions

Appendix: `with` and `return`

- When I say the `with` operator ensures the file is closed in *any* scenario, I do mean *any* scenario, and this includes returning.

```
readfile = None

def foo():
    global readfile
    try:
        with open("balance.txt") as readfile:
            print("Closed 1: {}".format(readfile.closed))
            return
            a = 5 / 0
            print("Closed 2: {}".format(readfile.closed))
        print("Closed 3: {}".format(readfile.closed))
    except:
        print("Caught!")
    print("Closed 4: {}".format(readfile.closed))

foo()
print("Closed 5: {}".format(readfile.closed))
```

with3.py

Output:

```
Closed 1: False
Closed 5: True
```

Appendix: `with` and Multiple Files

- The `with` operator can be used with more than one file as well.

```
with open("balance.txt") as readfile, open("poem.txt") as readfile2:
    print("File #1 closed 1: {}".format(readfile.closed))
    print("File #2 closed 1: {}".format(readfile2.closed))

print("File #1 closed 2: {}".format(readfile.closed))
print("File #2 closed 2: {}".format(readfile2.closed))
```

with4.py

Output:

```
File #1 closed 1: False
File #2 closed 1: False
File #1 closed 2: True
File #2 closed 2: True
```

Appendix: Opening the Same File for Reading Multiple Times at Once

```python
file = open("poem.txt")
print("=== File contents ===\n{}".format(file.read()))
file2 = open("poem.txt")
print("=== File contents ===\n{}".format(file2.read()))
file.close()
file2.close()
```

open-poem-twice.py

```
=== File contents ===
There once was a man from Peru;
who dreamed he was eating his shoe.
He woke with a fright
in the middle of the night
to find that his dream had come true.

=== File contents ===
There once was a man from Peru;
who dreamed he was eating his shoe.
He woke with a fright
in the middle of the night
to find that his dream had come true.
```

Appendix: Closing Twice Doesn't Raise Exception

- Closing a file twice does not raise an exception.

```
f = open("blah.txt")
print(f.read())
f.close()
f.close()

with open("blah.txt") as f:
    print(f.read())
    f.close()
```

double-close.py

```
AAA
AAA
```