File Input and Output, Handling Exceptions (Ch. 6)

CSC110: INTRO TO COMPUTER PROGRAMMING WITH PYTHON

INSTRUCTOR: BILL BARRY

1

R o a d m a p

☞ File I/O, Exceptions

Lists

List Algorithms

Dictionaries & Sets

Final Exam

2

This Week: File I/O, Exceptions

- ✓ Reading and Writing Files
- ✓ Using Loops to Process Files
- Exceptions, User Input Validation

Handling Exceptions, User Input Validation

3

Reading and Writing Files (aka File I/O)

4

Introduction to File Input and Output

Rationale: programs often need to save data that must *persist* beyond a single program run in memory; files are the usual way to store and retrieve that data

There are many **types of files**, including…

- Human-readable text files (we'll use these)
- Binary files that are less human readable and are often "dumps" of internal information

There are different **layouts** for data stored in a file, including…

- One item per line (read entire line into a variable)
- Multiple items on a line, delimited by spaces, commas (CSV), etc. (use split method)

5

Introduction to File Input and Output

There are two basic ways of **accessing** files:

- **Sequential**: read/write one thing after another, start to end (i.e., read the 1st record, 2nd record)
  - We'll use these
- **Random Access**: jump directly to where you want to go (e.g., the 43rd record); these require each record to be of fixed size. These are usually binary (not human readable, not simple text) files
  - We won't study these this quarter

6

Python Keywords

The main Python keywords we'll focus on. Please pay attention to which are functions and which are methods!

`open`	a **function** that opens a file and returns a file object
`close`	a **method** that closes an open file
`write`	a **method** that writes data to a file
`readline`	a **method** that reads one complete line from a file

The methods are called on a **File Object** (we'll get to that…)

7

File I/O in Python

Things we need when we open a file:

1. A **variable** to hold the File Object. We use this variable every time we do a file operation, and, later, to close the file when we're finished with it
2. The **name** of the file (technically, the file path); use this only *once*, when you open the file
3. A **specifier** indicating the *mode* we want to use:
   a) `'r'` **read** from an existing file
   b) `'w'` **write** into a new file (overwrites an existing file)
   c) `'a'` **append** (add) to an existing file (writes a new file if the file doesn't exist)

Command syntax: *fileVariable* = open(*filename*, *mode*)

8

File I/O in Python

Here are two examples of opening files in Python

We'll **w**rite to the patient file

```
patientFile = open('patients.txt', 'w')

hobbyFile = open(r'c:\MyFiles\myhobbies.txt', 'r')
```

We'll **r**ead from the hobbies file. Note the "**r**" before the file path, telling Python this is a "**r**aw" string; it won't interpret backslashes as escape characters

9

Notes on Text Files

We're going to use *plain text files* for our work

If you're creating these with a Python script, no worries

But if you're creating them from scratch, you must use a program that can work with plain text files, and not writing anything fancy into them

On **Windows**, use **NotePad**, or download **NotePad++** (a free, handy tool to have)

On the **Mac**, use **TextEdit**, but change its settings to Plain Text (it's not a file type, it's a setting), or download a program editor like **Brackets**

Also be careful of pasting text in from Word, a PDF, etc.; that can bring in unintended data like smart quotes (`‘hello’ vs. 'hello'`) and/or invisible characters that may cause trouble

10

Writing to a File

In general, to **write** to a file, one piece of data per line:

Patient file:

```
John Doe
43
…
```

```
patientFile = open('patients.txt', 'w')
patientFile.write(patientName + '\n')
patientFile.write(str(patientAge) + '\n')
patientFile.close()
```

Note that to append the newline character, patientAge must first be converted to a string, using the str() function

11

Reading from a File

In general, to **read** from a file, one item per line:

Hobby file:

```
Piano
23
…
```

```
hobbyFile = open('hobbies.txt', 'r')
hobbyName = hobbyFile.readline()
hobbyName = hobbyName.rstrip('\n')
hobbyYears = hobbyFile.readline()
hobbyYears = int(hobbyYears.rstrip('\n'))
hobbyFile.close()
```

Note that data comes in with **newline** character still attached; strip that.
Note that data comes in as **strings**; convert if you want numbers.

12

Pause & Practice

Write a function called inputAndWriteSchoolData that asks the user for their SID, name, and GPA, then writes them to a file called myschooldata.txt (one piece of data per line), then closes the file

Then, write readAndDisplaySchoolData that opens the file, reads the data back in and displays the data in this format shown below.

```
SID  : 123123123
Name : Chris Chavez
GPA  : 3.0
```

13

Pause & Practice: Code, #1

```
def inputAndWriteSchoolData():
    sid  = input('Enter your SID : ')
    name = input('Enter you name : ')
    gpa  = input('Enter your GPA : ')

    schoolFile = open('myschooldata.txt', 'w')
    schoolFile.write(sid  + '\n')
    schoolFile.write(name + '\n')
    schoolFile.write(gpa  + '\n')
    schoolFile.close()
```

14

Pause & Practice: Code, #2

```
def readAndDisplaySchoolData():
    schoolFile = open('myschooldata.txt', 'r')
    sid  =       schoolFile.readline().rstrip('\n')
    name =       schoolFile.readline().rstrip('\n')
    gpa  = float(schoolFile.readline().rstrip('\n'))
    schoolFile.close()

    print('SID  :', sid)
    print('Name :', name)
    print('GPA  :', format(gpa, '.1f'))
```

15

Using Loops to Process Files

16

Using Loops to Process Files

We can use loops to process files; most files contain multiple “records” of data

We’ll look at ways to use while or for loops to do this

We’re not storing the entire’s file contents as we’re reading in the data; we’ll learn that later. Because of this, we must do all processing needed *within* the loop

Testing note: always test your code by using an *empty* as well as *data-containing* file

17

Using `while` Loops to Process Files

We can use a while loop to read file contents:

```
hobbyFile = open(r'c:\JimsFiles\myhobbies.txt', 'r')
hobbyName = hobbyFile.readline()     # priming read
while hobbyName != '':               # empty string means EOF
    hobbyName = hobbyName.rstrip('\n')
    hobbyYears = hobbyFile.readline()
    hobbyYears = int(hobbyYears.rstrip('\n'))
    print(hobbyName, hobbyYears)
    hobbyName = hobbyFile.readline()
hobbyFile.close()
```

18

Hobby file:

```
Piano,5
Reading,13
```

Using `for` Loops to Process Files

We can use a for loop to read file contents, too; Python treats file as *collections* of lines of data (cool!)

```
hobbyFile = open(r'c:\JimsFiles\myhobbies.txt', 'r')
for hobbyLine in hobbyFile:
    print(hobbyLine)
hobbyFile.close()
```

This code reads in entire lines (“records”) at a time; you must split out into pieces (“fields”) if needed

19

Pause & Practice

Update your student GPA input/write code to allow the user to enter data for multiple students. Either use a “sentinel” to stop input when a certain SID is entered (e.g., “999-99-9999”) or ask them each time whether they want to enter another student

If time, update the read/display code to read in all student data and display it. Remember that you get *one* attempted read at the end of the file; if you try to continue reading, you’ll get an error. To solve this, consider what I call an *Early Escape* loop:

```
open file
read field #1
while field #1 contents aren't an empty string
    read field #2, etc.
    do the rest of the stuff
    read field #1 again
close file, do wrap-up stuff
```

20

Pause & Practice: Input/Write

```
def inputAndWriteSchoolData():
    schoolFile = open('myschooldata.txt', 'w')

    wantAnother = True
    while wantAnother:
        sid  = input('Enter your SID : ')
        name = input('Enter you name : ')
        gpa  = input('Enter your GPA : ')

        schoolFile.write(sid  + '\n' + name + '\n' + gpa  + '\n')

        another = input("Enter another (Y/N): ")
        if another != "Y":
            wantAnother = False

    schoolFile.close()
```

21

Pause & Practice: Read/Display

```
def readAndDisplaySchoolData():
    schoolFile = open('myschooldata.txt', 'r')
    sid = schoolFile.readline().rstrip('\n')
    while sid != "":
        name =       schoolFile.readline().rstrip('\n')
        gpa  = float(schoolFile.readline().rstrip('\n'))

        print('SID  : ' + sid)
        print('Name : ' + name)
        print('GPA  : ' + format(gpa, '.1f'))

        sid = schoolFile.readline().rstrip('\n')

    schoolFile.close()
```

22

Handling Exceptions, User Input Validation

23

Exception Analogy

Toddlers learning to walk will sometimes fall and scare or hurt themselves; parents know this

To keep bad things from happening, parents often walk just behind/above the child, so that if the child loses balance the parent can prevent the fall

If the child doesn't lose their balance, then the parent doesn't need to act; the child continues, unaware

If the child starts to fall, the parent jumps into action and handles the issue

24

Exception Analogy, continued

To get a bit more technical…

The toddler learning to walk is engaged in a **risky operation**

The parent hovers while the child will **try** to walk

The parent **catches** the child if it starts to fall; we could also say the parent **handles** the bad thing that happens, so there's not a catastrophe

25

Handling Exceptions

At program runtime, things often go wrong and blow up your code. Examples:

Users don't enter the data you want them to, e.g., they enter "forty-three" when you want 43

Files aren't where you expected them to be, or have been deleted

Values generate unexpected results, like num1/num2 when num2 is 0

26

Exception Handling in Python

```
try:
    # potentially risky operations
except:
    # code to handle specific problem, if it arises
else:
    # code to run only if no exceptions are caught
finally:
    # code to run regardless of whether there were exceptions
```

Note: the `else` and `finally` sections are optional; often you see only try/except

27

Exceptions: A Simple Example

Looking at the code below, what could go wrong?

```
age = int(input("Enter your age: "))
dogYears = age // 7
print("In dog years, that is", dogYears)
```

Let’s use Exception Handling to solve the data input problem

28

Exception Handling, Step #1

The first step is to know the exact exception we get when the bad thing happens; copy it as you'll need it soon

```
Enter your age: 29.5
Traceback (most recent call last):
  File "C:\Users\Bill\Desktop\Hello.py", line 1, in
<module>
    age = int(input("Enter your age: "))
ValueError: invalid literal for int() with base 10: '29.5'
```

29

Exception Handling: Step #2

Next, we put the "risky" code in a try block:

```
try:
    age = int(input("Enter your age: "))

dogYears = age // 7
print("In dog years, that is", dogYears)
```

30

Exception Handling: Step #3

Next, we catch/handle the specific exception using the except block:

```
try:
    age = int(input("Enter your age: "))

except ValueError:
    print("Age must be an integer; terminating")

dogYears = age // 7
print("In dog years, that is", dogYears)
```

31

Exception Handling: Step #4, Option a

Next, we continue “business as usual” in the else block:

```
try:
    age = int(input("Enter your age: "))
except ValueError:
    print("Age must be an integer; terminating")
else:
    dogYears = age // 7
    print("In dog years, that is", dogYears)
```

32

Exception Handling: Step #4, Option b

Another option: “business as usual” goes after risky code (not in else); the benefit is that the exception handling is at the very bottom

```
try:
    age = int(input("Enter your age: "))
    dogYears = age // 7
    print("In dog years, that is", dogYears)
except ValueError:
    print("Age must be an integer; terminating")
```

33

Exception Handling: Step #5

Now, we test it in both the “business as usual” and the “bad thing” cases:

```
Enter your age: 29
In dog years, that is 4 ✓
>>>
Enter your age: 29.5
Age must be an integer; terminating ✓
>>>
```

34

Input Validation Loop

```
validAge = False      # assume bad data; gets into loop
while not validAge:   # keep looping while data is bad
    try:
        age = int(input("Enter your age: "))
    except ValueError:
        print("That was not integer; please try again\n")
    else:
        validAge = True

dogYears = age // 7
print("In dog years, that is", dogYears)
```

35

Moving Input Validation to a Function

Let’s move the “get an integer from the user” code into a function called **getInteger**; we know we’ll need it in this or other programs

Putting the resulting function into our own little Utility.py module might not be a bad idea, either; that way we can import and use it in any program we write

36

getInteger Function

```
def getInteger(promptText):
    validInt = False
    while not validInt:
        try:
            userInt = int(input(promptText))
        except ValueError:
            print("That was not integer; please try again\n")
        else:
            validInt = True
    return userInt
```

37

Using getInteger in Code

```
def main():
    age = getInteger("Enter your age: ")
    dogYears = age // 7
    print("In dog years, that is", dogYears)

main()
```

38

Sample Run

```
Enter your age: 29.5
That was not integer; please try again

Enter your age: 32.3
That was not integer; please try again

Enter your age: 31
In dog years, that is 4
```

39

Exception Notes

You can handle more than one exception, and even handle “anything else” that might happen:

```
try:
    # risky stuff
except IOError:
    # …
except ValueError:
    # …
except:
    # …
```

40

The End

41