Files and Errors

- **Interacting with Files**
- **File Access**
- **Reading from Text Files**
- **Writing to Text Files**
- **Reading and Writing Complex Data Files**
- **Exception Handling**
- **Different Techniques**
- **Reading:**
  - Dawson, Chapter 7
  - http://introcs.cs.princeton.edu/python/15inout

Interacting with Files

- So far, we have been writing our programs in a way that we have directly supply our data. The data is either…
  - Defined within the code itself (i.e., stored in variables), or…
  - …generated during code execution (for example, from user input)
- Regardless, the data disappears after the program quits.
- For our purposes, this has not usually been a problem because our programs have not needed to keep data between program runs

Interacting with Files

- However, for most programs to be useful, they need to be able to load and unload data from records.
- This quality – where data is taken from a source when the program starts and the source updated before termination – can be considered a form of *data persistence*.
- One important way of keeping persistent data is to save it in <u>files</u>. This provides a number of important benefits:
  - Allows us to maintain records of changeable information
  - Further emphasizes the separation of data from program logic.
  - Some tasks can be automated – or at least made more efficient – by providing the required information in data files.

File Access

- Here, we will see this done with both plain text files (usual extension: **`.txt`**) and binary data files (one extension: **`.dat`**)
- Interacting with plain text files will be much simpler and provide the added benefit of being able to read the data directly by opening the file in a text editor.
- For interacting with a text file, you will need to know where it is located on your system – in other words, which folder – so that you can direct Python to it.
- This piece of information is called a *file path*. It can be expressed in either of two forms…

File Access

- **Absolute path:** Consists of the full location, all the way up to and including the main drive. *Examples*:
  - ***Windows:*** *`C:\Users\John\Documents\Classes\CS110\file.txt`*
  - ***Linux:*** *`/home/john/documents/classes/cs110/file.txt`*
- **Relative:** Where the path is expressed according to its location relative to the current directory. *Examples* (assume your current directory is a sibling directory of “Classes”/“classes”):
  - ***Windows:*** *`..\Classes\CS110\file.txt`*
  - ***Linux:*** *`../classes/cs110/file.txt`*
- For the relative file paths, the two dots **..** mean "the parent directory of the current".

File Access

- Similarly, you could use a single dot **.** to indicate "the current directory"
- Notice the difference between file path styles in Windows versus a Linux system.
  - Windows uses ***back-slashes*** as separators, while Linux uses ***forward-slashes***.
  - When you supply a file path to Python, you will provide it in *string* form.
  - Since back-slashes are special characters in a string, you will need to escape it using *another* back-slash.

File Access

- As such, a file path in string form might look something like this:
  `"C:\\Users\\Documents\\John\\Classes\\CS110\\file.txt"`
- This should not be an issue when providing a Linux-style file path

- If you decide to use relative file paths, consider that when Python is running a program, the "current directory" is usually the one where the code file is located.
- Therefore, the file path must be relative to that directory.
- If the file is in the same folder, then you need only provide the file name itself. *Example*: **`"file.txt"`**

File Access

- Once you do this, you will have a number of options for opening the file for reading or writing.
- The simplest way to open a file is the **open** function, with the file path as a parameter.

```
my_file = open ("file.txt")
```

- You may also specify a parameter to open the file *for a specific purpose*. This, for example, will open the file for *reading*:

```
my_file = open ("file.txt", "r")
```

File Access

- You may see other such parameters in the textbook and in code examples.
- The **open** function is a Python built-in, and it returns to you a *file object*. This object will have various functions related to reading from and writing to *that file*.
- In addition to completing Homework 10, you may also find it helpful to simply experiment with file objects and their functions.

Reading from Text Files

- Your file object will offer a number of functions and techniques for reading the file data.
- The simplest will be the **read** function, using no parameters, which returns the entire file text as one large string object

```
file_text = my_file.read()
```

- In class, you may have seen me execute code in Sublime Text in the following manner:

```
exec (open ("code_file.py").read())
```

Reading from Text Files

- The **open** function opens the file for reading, the **read** function returns the code as a large multi-line string, and the **exec** function interprets the string contents as Python code – just like it would code you type in at the command line.
- In addition, you may read the file text by one character or more at a time. The following sequence of commands….

```
print (my_file.read (5))
print (my_file.read (7))
print (my_file.read (5))
```

- …will read and print the first five characters, followed by the next seven, followed by the next five. Here, the read function returns a sequence of characters as a string.
- The file object remembers your "place" in reading

Reading from Text Files

- You may also read the file line by line, getting each line as a string. The following code will print the next five lines of the file:

```
for i in range (5):
    print (my_file.readline())
```

- In addition, you may read the line text by one character or more at a time. The following sequence of commands….

```
print (my_file.readline (5))
print (my_file.readline (7))
print (my_file.readline (5))
```

- …will read and print the first five characters of the current line, followed by the next seven, followed by the next five. Again, the read function returns the characters as a string.

Reading from Text Files

- As before, the file object remembers your "place" in reading
- If you read a number of characters ***larger*** than there are remaining in the line, then it will simply give you back the rest of the current line – but none of the next!
- If there is no line left in the file to read, then the **readline** function will return an empty string.
- Finally, you can use the **readlines** function to get a list of all the (remaining) lines in the file. The following sequence of commands….

```
for line in my_file.readlines():
    print(line)
```

- …will read and print the rest of the lines in the file.

Reading from Text Files

- An alternative to the above syntax:

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

- In this case, it is as if you are "looping through the file"
- Don't forget to close the file afterward: **`my_file.close()`**
- This is important for at least two reasons:
  - On one hand, it is simply good programming practice. If you open a file, then you should remember to close it.
  - Also, closing a file makes it possible for you to <u>*re-open*</u> it and start on it from the beginning.
- This will also apply to writing files.

Writing to Text Files

- Similarly, file object also offers some functions and techniques for writing to a file.
- To start with, you will have to open a file for writing. You can do that like this:

```
out_file = open ("out_file.txt", "w")
```

- If the file does not exist, then it will be created. If it does, then it is replaced by a new, empty file of the same name.
- The simplest way to write to the file will be the **write** function, using a string of text as your parameter.

Writing to Text Files

- For example:

```
out_file.write("This is some text\n")
```

- Notice the inclusion of the newline character. This is because the <u>**write**</u> function writes the exact characters specified by you, into the file.
- If you do not include the newline characters yourself, then all the text will be written to a single line.
- As with reading, the file object will remember your place in writing. Any further text written to the file will begin where the last text ended.

Writing to Text Files

- The following sequence of statements….

```
out_file.write("This is line #1\n")
out_file.write("This is line #2\n")
out_file.write("This is line #3\n\n")
out_file.write("This is the last line...")
out_file.close()
```

- …executed on an empty file, leaves the file looking like this:

```
This is line #1
This is line #2
This is line #3

This is the last line...
```

Writing to Text Files

- Also, the **writelines** function takes a *list* of strings as a parameter and writes them all to the file. For example:

```
lines = ["This is line #1\n",
         "This is line #2\n",
         "This is line #3\n\n",
         "This is the last line..."]
```

- If `out_file` is empty, then these statements...

```
out_file.writelines(lines)
out_file.close()
```

Writing to Text Files

- ...will leave the file looking like this:

```
This is line #1
This is line #2
This is line #3

This is the last line...
```

- As with reading a file, you should *close* the file once you are finished with it.
- For more ways to open a file, check out Table 7.1 on page 193 of the textbook. To ***add*** text to an existing file, rather than overwriting it, use the **"a"** option when opening the file.

Reading and Writing Binary Data Files

- Text files can be very useful as a basic way of storing data in files and retrieving it later.
- However, when that data requires more complex operations to be translated to and from file text, this will become more burdensome.
- For example, it might involve interpreting and transforming the data.
- Fortunately, Python offers us at least two ways to preserve data in files without having to worry about the smaller details.
- Here, we will look at *pickling* and *shelving*. As you will see, these techniques store data in **`.dat`** files, rather than **`.txt`**
- As such, you will usually not be able to open and read **`.dat`** files as you would **`.txt`** files.

Pickling

- The term is a reference to the process of preserving food via either submersion in vinegar or fermentation in salt water.
- In Python, pickling entails preserving a data object by writing the object itself to a file, so that it can be retrieved later.
- To start with, you will have to import the **pickle** module:

```
import pickle
```

- Also, you should have a data object, such as a list or dictionary
- Then, you must open a file for *binary* writing, which entails a variation on our previous use of the **open** function:

```
storage = open ("storage_file.dat", "wb")
```

- This will open the file for writing data in *binary* form.

Pickling

- From here onwards, the code will be different from what we've seen before
- To pickle an object, you will need to use the **pickle.dump** function
- Let's say you have a list...

```
sweets = [ "cookies", "pies", "cakes", "muffins",
"candy" ]
```

- ...and you want to pickle it. With the previously declared variables, you would use a statement like this:

```
pickle.dump (sweets, storage)
storage.close()
```

- As before, you should **close** a file explicitly, after you are done with it.

Pickling

- Later, if you wanted to use that same list in this or another program, you would use the **pickle.load** function to retrieve the data (*unpickle* it) as a list object:

```
storage = open ("storage_file.dat", "rb")
```

- This will open the file for *reading* data in binary form. Then, you could fetch the next object stored in the file:

```
desserts = pickle.load (storage)
```

- This, of course, requires *knowing in advance* the order in which the data objects were added to the file, in the first place.
- See Table 7.4 on page 203
- Many different types of objects can be pickled and unpickled. Experiment with some and see what happens.

Shelving

- Next, we have *shelving*, which is like pickling -- but a bit more sophisticated.
- To use shelving, your program must first import the shelve module:
  ```
  import shelve
  ```
- It works with a shelf object, which is based on a **`.dat`** file.
- To get this object, you would use a statement like the following:
  ```
  shelf = shelve.open("shelf_file.dat", "c")
  ```
- The parameter **`"c"`** indicates that, if the file does not already exist, then it should be *created*. Though we will not go into it here, see Table 7.5 on page 204 for more access modes.

Shelving

- The variable **`shelf`** will now contain a shelf object, which you can interact with much as you would a dictionary object:

```
shelf["desserts"] = [ "cookies", "pies", "cakes", "muffins",
"candy" ]
shelf["fruits"] = [ "apples", "oranges", "bananas" ]
shelf["beverages"] = [ "water", "coffee", "tea" ]
shelf.sync()
```

- Here, the **sync** function updates the data file so that it reflects the latest changes to the shelf object data.
- When you want to retrieve the data, you would use the shelf object and the appropriate key -- again, just as you would with a dictionary.

```
print ("Desserts:", shelf["desserts"])
print ("Fruit options:", shelf["fruits"])
print ("Drink options:", shelf["beverages"])
```

Shelving

- When you are finished with the file, be sure to close it:
  `shelf.close()`
- **close**, like **sync**, will update the file. When you go back and use the data file later, in the same or a different program, the data will be just as you left it.
- If you want to practice with this method of data storage, try revisiting some of our earlier examples using dictionaries and trying to re-implement them using shelving.
- See if you can make it work – and have the data persist between runnings of your program!

Exception Handling

- While a program is running, it may be presented with an error that makes further execution impossible. As such, the program will crash.
- This particular scenario is called an *exception*. This is because the error is an exceptional situation that the Python interpreter cannot handle.
- When it happens, we say that the offending code "raises" an exception.
- Examples:
  - Dividing by zero
  - Attempting to concatenate a string to a number
  - Attempting to convert a non-numeric string to a number
  - An expression where the operator and data types are incompatible

Exception Handling

- In a compiled language, such as Java, some of these would not cause exceptions because they would prevent the code from compiling in the first place.
- Up until this point, in our programs, if an exception is raised, then the program would simply crash. This is not something that you want to happen in professionally-made software.
- However, another option we have is to *handle* them.
- "Handling" an exception means that, in the event of an error, we can have the program respond to it and continue -- or, at least, quit naturally-- instead of crashing.

Exception Handling

- This works in two parts:
  1. We *try* to execute a series of statements.
  2. If an exception arises, then we *handle* it with alternative code.
- We do this using a **try statement** with an **except clause.** This code would take the following form:

```
try:
    x = math.sqrt(-9.0)
except:
    print ("We cannot use math.sqrt in this way!")
```

- The **try** component contains the code that we *want* to execute. This is code that we know could potentially create an error during runtime.

Exception Handling

- If an error arises, we will "trap" or "catch" it -- where it is handled by the next part...
- The **<u>except</u>** clause contains the code that we will execute in response, if an error is raised while the **<u>try</u>** code is executing.
- If the **<u>try</u>** code completes without error, then the **<u>except</u>** code is skipped entirely.
- Regardless, usual program execution will resume after one of these is finished.
- In addition, your **<u>except</u>** clause can specify the *<u>kind</u>* of exception it is intended to handle.

Exception Handling

- Here, we catch and handle a <u>ValueError</u>:

```
try:
    x = math.sqrt(-9.0)
except ValueError:
    print ("We cannot use math.sqrt in this way!")
```

- If your **<u>except</u>** clause is specific in this way, then other types of errors -- by going uncaught -- could still cause a program crash here. Example:

```
try:
    x = int (input ("Please enter an integer: "))
    result = 1 / x
except ValueError:
    print ("Cannot convert that value to an int!")
```

Exception Handling

- If the user enters something that cannot be converted to a integer value, then the **except** code will execute. However, if the user enters a value of *zero*, then the program would still crash, because that is a different kind of error, called a ZeroDivisionError
- (For some common exception types, see Table 7.6 on page 207.)
- There are some other variations on the "try-except" structure.
- For example, you can have *multiple except clauses*:

```
try:
    x = int (input ("Please enter an integer: "))
    result = 1 / x
except ValueError:
    print ("Cannot convert that value to an int!")
except ZeroDivisionError:
    print ("Cannot divide by zero!")
```

Exception Handling

- This way, *either* type of error will be caught and handled.
- Also, one except clause can handle more than one error type:

```
try:
    x = int (input ("Please enter an integer: "))
    result = 1 / x
except (ValueError, ZeroDivisionError):
    print ("Oops! Either you entered a non-integer 
value or tried to divide by zero!")
```

- There are some kinds of errors that you will want to handle differently, and other kinds that you will want to handle in the same manner.
- Sometimes, in your **except** code, you may want to use the error message in some way.

Exception Handling

- As a matter of fact, when an error arises, an *exception object* is created, which you can assign to a variable to be used in the **except** code:

```
try:
    x = math.sqrt(-9.0)
except ValueError as exc:
    print ("We cannot use math.sqrt in this way! ")
    print ("Here is Python's error message: ")
    print (exc)
```

- Finally, you can also add an **else** clause after the **except** clause. This way, if the **try** code finishes executing *completely* and successfully, then this code will execute.

Exception Handling

- Here, for example, we include an **else** clause...

```
try:
    x = int (input ("Please enter an integer: "))
    result = 1 / x
except (ValueError, ZeroDivisionError):
    print ("Oops! Either you entered a non-integer
value or tried to divide by zero!")
else:
    print ("You gave a valid input -- good job!")
```

- The **else** clause could be anything that we want to see happen, if and when the **try** code successfully completes.

Exception Handling

- Exception handling is an important tool for preventing errors from crashing your code.
- It is not the only way. Sometimes, you will want to simply write the code so that the error cannot happen, in the first place.
- Other times, though, exception handling will be the way to go, particularly in situations where:
  - Something could go wrong -- but probably will not...
  - ...and catching/handling the exception would be more efficient than trying to prevent it
- Like many things, it is one of several options available to you.