File Input/Output

Up until now, input has been provided for you to hard-code into your program or it showed up in the form of user input (with `input()`). Now a new form of input (and output) is introduced: files! The most basic file type that you probably end up using will be a text file. It just stores text. These files have the `.txt` extension. An example is the "example.txt" file in the same folder as these notes. You can theoretically access and use any file type: you just need to ensure that your code has the correct support for the file extension you want to use.

Do note that Python can open pretty much every file natively, but it does not necessarily mean that it will read the data correctly. For example, `csv` files can be read normally with basic Python, but it is more ideal to open `csv` files with the creatively-named `csv` module. The `csv` module will not be tested: I only reference it for your information. If you wish to try it out, do so as you like but do not do so expecting to be tested on it. In fact, it may be *banned* from usage as it overly simplifies certain types of problems.

File Object

We need some way of storing the contents and properties of an entire file, which can be *any* file, into a variable that Python can then read and process. In simple terms, Python has its own data type for files, and you can store a file as something we call a file object. This special object is something you can interact with to tell Python to do stuff with your file data. The actual specifics of what a file object, or any object, really is are not necessary to know. You'll run into objects a lot more once you get into what we call "classes" and eventually what is called "object-orientated programming" (OOP). However, I will go over the basics of what you can do with this file object in the rest of these notes.

Opening and Closing a File

Consider a file object to be a physical book. Reading this physical book typically requires the following steps:

- Making the book (making a file object using a pre-existing file)
- Storing the book (storing the object in some variable)
- Grabbing the book (using the actual variable)
- Opening the book (getting access to file contents)
- Reading the book (doing whatever you want with the contents)
- Closing the book (closing the file)

It's a lot of steps! But not that many lines of code thankfully (other than the reading part). Example 11-1 will show the full process, and I'll explain each segment of code afterwards.

In [ ]:
```
# Example 11-1: Opening, accessing, and closing the file

file_object = open("example.txt", "r")

lines = file_object.readlines()
for line in lines:
    print(line)

file_object.close()
```

```
This

file

has

been

opened
```

Using open()

Line 3 from Example 1:

```
file_object = open("example.txt", "r")
```

So this covers the first three bullet points (making the book, storing the book, and opening the book). First, let's talk about `open()`. `open()` is a built-in function that lets you create a file object given a file directory and the access mode. By default, Python will look for files in the same folder as your code. You may notice that this notebook is in the same folder ("Module 11 - Files") as the text file ("example.txt"). This means that I can just tell Python to directly look for the file. Suppose the file directory looked like this though:

```
> Module 11 - Files
  > Files
    - example.txt
  - module_eleven_notes.ipynb
```

The text file is no longer in the same folder. My directory must tell Python where to navigate *starting* in the folder the file I am running is currently in. So, I am in "Module 11 - Files" and thus, I'll navigate from there. I need to first access the "Files" folder and then I get to

"examples.txt", so my directory becomes `"Files/examples.txt"`. When in doubt, just place your file in the same folder as your code or use absolute path (tells Python where to go starting at the very root of your file directory, so it doesn't matter where your code is).

Access Modes

Access modes tell Python how exactly you want to access the file. There are several options, with the important ones being:

- `r` : read-only mode
- `a` : append-only mode
- `w` : write-only mode

There are three other modes:

- `r+` : read + write mode
- `a+` : append + read mode
- `w+` : write + read mode

Read only mode means that you can only read the file. You can access the contents, but you cannot overwrite, remove, or add any data. Append only mode means that you just want to add data to the end of the file. Write only mode means that the entire file is cleared and you can add to the file just like append. A second thing that write only mode does is that it will also create the file if it does not exist.

Append + read mode is neat where you can read the file *and* add data to the end. However, `r+` and `w+` can be a bit confusing since they both allow for reading and writing. Note that `r+` has `r` in its name, which means read mode is its priority. The file will not be wiped (which write mode usually does), but in this case, you start writing data from the top of the file, overwriting any lines you write to. `w+` will clear the file first and allow you to write to the file, but you can still read the file contents. The `+` modes are rarely used, but I mention them for the sake of completion. I recommend you look up examples and play around with the modes by yourself to fully understand the modes.

Now specifically for the code from Example 1, the file is opened in read only mode due to the `"r"` passed into `open()`. `open()` itself is the act of opening the book.

Accessing file contents (readlines())

Lines 5-7 from Example 1:

```
lines = file_object.readlines()
for line in lines:
    print(line)
```

This code snippet demonstrates the fourth and fifth bullets (grabbing the book and reading the book). Line 5 uses a function called `readlines()`. This returns a list of all the lines (see Example 11-2). Several functions will be covered for accessing the file contents (reading the book). Note that I have to call the function on the file object: which means I have to get the file object (grab the book) before I can access the contents (read the book).

In [ ]:
```
# Example 11-2: readlines()

file_object = open("example.txt", "r")

print(file_object.readlines())

file_object.close()
```

```
['This\n', 'file\n', 'has\n', 'been\n', 'opened']
```

Note that even the newline characters are stored in the line. Sometimes, the ending will be `\r\n`, with `\r` called a carriage return. The details of what `\r` does is not relevant to this course, but its existence does. When you are reading these lines, you need to filter out the potential `\r`s and `\n`s so you get just the data you want. The easiest way to do this is with `strip()`:

In [ ]:
```
# Example 11-3: Stripping a line

line = " \r\nHello\r\n"

print(repr(line))
print(repr(line.strip()))
```

```
' \r\nHello\r\n'
'Hello'
```

You can see that with `strip()`, the special characters at the end are stripped (note that `strip()` will also remove whitespace and it works on *both* ends of the string). `repr()` is a built-in function that shows the raw string (which means special characters will be outputted instead of processed). As you can see, even though `line` has a `\n`, it still appears in the output, and there is not an additional newline. This is an extremely strong function to use to check the actual contents of strings, which can be very helpful if you have mysterious blank lines or line processing mishaps due to the normally invisible special characters.

With `readlines()`, you can just store the list in a variable and loop through that list as you need to. But what if you want to read line-by-line and not access everything at once?

Accessing file contents (readline())

Neatly enough, you can just request a line.

In [ ]:
```
# Example 11-4: readline()

file_object = open("example.txt", "r")

print(file_object.readline())

file_object.close()
```

```
This
```

It's not my personal favorite to use since there are more elegant ways for me to read through a whole file than to call `readline()` over and over again, but it does have its time and place. If you know exactly how many lines you need, you can use a `for` loop and grab an exact amount of lines.

Accessing file contents (read())

`read()` is very strong since it stores *the entire file into one string*.

In [ ]:
```
# Example 11-5: read()

file_object = open("example.txt", "r")

print(repr(file_object.read()))

file_object.close()
```

```
'This\nfile\nhas\nbeen\nopened'
```

This is usually the alternative to `readlines()`, especially if there's a common pattern or lines are grouped together and it's easier to `split()` the string. `split()` will be discussed later in these notes but keep note of what it does in Example 11-6.

In [ ]:
```
# Example 11-6: Splitting the read()

file_object = open("example.txt", "r")

print(file_object.read().split("\n"))

file_object.close()
```

```
['This', 'file', 'has', 'been', 'opened']
```

Looping over the object

Weirdly enough, you can treat the file object as an actual list of lines. Now of course, this is not entirely accurate, but for the sake of the course, it's accurate *enough* for our needs. Example 6 highlights using a `for` loop to iterate through the file object itself:

In [ ]:
```
# Example 11-7: Iterating over the file object

file_object = open("example.txt", "r")

for line in file_object:
    print(line)

file_object.close()
```

```
This

file

has

been

opened
```

Although nearly identical to using `readlines()`, a major difference is that you do not store the lines as you go through the loop. You could circumvent this by making a list and appending each line into the list, but that's essentially `readlines()`. The use case differences are generally too small to consider (at least for this class), so it's up to personal preference. Sometimes I prefer `readlines()` when I know I want the whole list of lines immediately to do something to the strings, and technically doing `for line in file_object` is more memory efficient (and also arguably is cleaner looking), but regardless, these decisions are not important for this class. Whether you choose to use `readlines()` or a `for` loop, all that matters is that you use them correctly.

Using close()

Note that in all my snippets, I used `close()` on the file object. Line 9 from Example 1:

`file_object.close()`

If I leave the file open, it opens opportunities for the file to be accidently be altered. To ensure the data stay safe and intact, I need to close the file (close the book). For an analogy, imagine that in real life, if I leave the book open, the inside pages are exposed to the outside and could be damaged by energetic pets or spilling a drink. Thus, I must close the book and return it (which Python will automatically do in the background after you `close()`) to make sure the book is as safe and intact as possible.

Technically, you don't *need* to `close()` (Python won't throw an error), but it is extremely good practice to do so. The negative consequences of leaving a file open far outweighs the benefit of the time saved of skipping one line of code.

Using" with"

There's a way for you to open a file and have Python automatically close the file for you. This is possible through the `with` keyword. Example 11-8 shows how you can rewrite Example 1.

In [ ]:
```
# Example 11-8: A cleaner Example 1

with open("example.txt", "r") as file_object:
    lines = file_object.readlines()
    for line in lines:
        print(line)
```

```
This

file

has

been

opened
```

The `with` keyword is super powerful in that it opens *and* closes the file for you. Once Python is completely done executing the code within the `with` block, Python will close the file for you without you needing to explicitly call `close()`. Using `with` thus is the preferred way to open files in this class.

split()

`split()` lets you split a string into a list of strings. Although this doesn't sound super good, it can be incredibly powerful whenever you need to extract information from strings (which happens quite a lot). To determine what to split the string with, `split()` needs to know what the *delimiter* is: a sub-string that used to visually separate the data in the string:

`string = "Hi|Yo|Bye"`

From the above line, you can expect `"|"` to be the delimiter since it is separating the words. If I wanted to split that string into a list of strings using `"|"` as the delimiter, then I would do it as shown in Example 11-9:

In [ ]:
```
# Example 11-9: Splitting the string

string = "Hi|Yo|Bye"

print(string.split("|"))
```

```
['Hi', 'Yo', 'Bye']
```

Note that the delimiter itself is not included. It's similar to cutting a piece of paper in half: you can fold the paper in half or mark a line down the middle then use that fold or mark as a reference of where to cut. After you cut the piece of paper, the fold/mark disappears. Note that if you want to just split on whitespace, then you can `split()` with no argument provided:

In [ ]:
```
# Example 11-10: Split without delimiter

string = "Tony Alana\nSam\rLuke"

print(string.split())
```

```
['Tony', 'Alana', 'Sam', 'Luke']
```

join()

Unsuprisingly, there's a function to combine a list of strings using a delimiter. This function is called `join()`. Note that the list you are joining must all be strings. So if you want to combine a list of numbers, you need to first cast each number into a string and then you can join them together. Also note that `join()` is actually *called on the delimiter string* and *takes in the list as its argument*:

In [ ]:
```
# Example 11-11: join()

names = ["Tony", "Anthony", "Big T"]

print(", ".join(names))
```

```
Tony, Anthony, Big T
```

CSV Files

CSV (comma-separated values) is a special type of text file. You can open and read this file exactly like a regular text file, but you can use the fact that this file has values separated by commas to `split()` and extract data. Typically, these files are primitive data tables. This means that processing a CSV file is really not that different than processing a text file: you just need an extra step of splitting.

In [ ]:
```
# Example 11-12: Processing a CSV file

with open("example.csv", "r") as csv_file:
    for line in csv_file:
        print(line.split(","))
```

```
['T', 'To', 'Ton', 'Tony\n']
['L', 'Lu', 'Luk', 'Luke\n']
['A', 'Al', 'Ala', 'Alan', 'Alana\n']
['H', 'Ho', 'Hoi', 'Hoid']
```

In the above, there are newlines still present in the last string of each line except the last! This can really mess up your string processing since that newline character is part of the string and can lead to some weird behaviors or even errors depending on what you plan to do with the string. This means that you should strip the string. **In general, if you are processing lines from a file, consider stripping the line**.

In [ ]:
```
# Example 11-13: Stripping before processing

with open("example.csv", "r") as csv_file:
    for line in csv_file:
        print(line.strip().split(","))
```

```
['T', 'To', 'Ton', 'Tony']
['L', 'Lu', 'Luk', 'Luke']
['A', 'Al', 'Ala', 'Alan', 'Alana']
['H', 'Ho', 'Hoi', 'Hoid']
```

Append Mode

I left this section (along with write mode which will be after this section) at the end because I wanted to introduce tools such as `split()` and `join()`, which are really good for processing lines and creating strings. After splitting then joining, or just joining, you can reshape data to what you want and then add to your file. Append mode is `"a"` and it only adds lines to the end of the file. Let's add some more names to the csv file:

In [ ]:
```
# Example 11-14: Appending to a file

with open("example.csv", "a") as csv_file:
    names = ["Luna", "Gabe", "Andrew"]

    # Form the cool names
    for name in names:
        name_partials = []

        # Start with first char, then first two, etc.
        # T, To, Ton, Tony
        for i in range(len(name)):
            name_partial = name[:i+1]
            name_partials.append(name_partial)

        # Join and append the list
        new_line = ",".join(name_partials)
        csv_file.write(new_line + "\n")   # Newline!!!!

# Check contents
with open("example.csv", "r") as csv_file:
    for line in csv_file:
        print(repr(line))
```

```
'T,To,Ton,Tony\n'
'L,Lu,Luk,Luke\n'
'A,Al,Ala,Alan,Alana\n'
'L,Lu,Lun,Luna\n'
'G,Ga,Gab,Gabe\n'
'A,An,And,Andr,Andre,Andrew\n'
```

Two key things to note:

- Appending to a file uses `write()` (it's the same function used in write mode!)
- I had to add a newline to the end of each line

`write()` is not the same as `print()` where `print()` will automatically go to the next line. If you want to write lines individually, you need to remember to add a newline to the end of each line so that the next line being written will actually be on the next line.

Write Mode

Write mode clears the file and then starts adding stuff to the file. It also can create the file for you if Python cannot find the file you specified. I won't make a new file, but I will demonstrate the file being cleared and then writing to it. Remember that to put a file in write mode, you need to use `"w"` as the access mode.

In [ ]:
```
# Example 11-15: Write mode

with open("example.txt", "w+") as f:
    # Demonstrate file is empty
    print("Original file contents:", repr(f.read()))

    # Rewrite to the file
    lines = "This file has been opened".split()
    for line in lines:
        f.write(line + "\n")

# Check contents
with open("example.txt", "r") as f:
    print("New file contents:")
    for line in f:
        print(repr(line))
```

```
Original file contents: ''
New file contents:
'This\n'
'file\n'
'has\n'
'been\n'
'opened\n'
```

I used `"w+"` so I can also have access to read mode, which I need to have to use `read()`. But the primary access mode is `"w"` since I used `"w+"` instead of `"r+"`, so Example 11-15 does still show write mode. You can remove the first `print` statement, and the code will still work after changing `"w+"` to `"w"`.

I use `repr()` here to show the raw string representation. You can see that after I open the file, there is nothing in the file as represented by the empty string. After writing to the file, then I have all five new lines inside the file.