CS50's Introduction to Programming with Python

OpenCourseWare

Donate (https://cs50.harvard.edu/donate)

David J. Malan (https://cs.harvard.edu/malan/)
malan@harvard.edu
(https://www.facebook.com/dmalan) (https://github.com/dmalan)
(https://www.instagram.com/davidjmalan/) (https://www.linkedin.com/in/malan/)
(https://www.reddit.com/user/davidjmalan)
(https://www.threads.net/@davidjmalan) (https://twitter.com/davidjmalan)

Lecture 6

File I/O

- Up until now, everything we've programmed has stored information in memory. That is, once the program is ended, all information gathered from the user or generated by the program is lost.
- File I/O is the ability of a program to take a file as input or create a file as output.
- To begin, in the terminal window type `code names.py` and code as follows:

```
name = input("What's your name?" )
print(f"hello, {name}")
```

Notice that running this code has the desired output. The user can input a name. The output is as expected.

- However, what if we wanted to allow multiple names to be inputted? How might we achieve this? Recall that a `list` is a data structure that allows us to store multiple values into a single variable. Code as follows:

```
names = []

for _ in range(3):
    name = input("What's your name?" )
    names.append(name)
```

Notice that the user will be prompted three times for input. The `append` method is used to add the `name` to our `names` list.

- This code could be simplified to the following:

```
names = []

for _ in range(3):
    names.append(input("What's your name?" ))
```

Notice that this has the same result as the prior block of code.

- Now, let's enable the ability to print the list of names as a sorted list. Code as follows:

```
names = []

for _ in range(3):
    names.append(input("What's your name?" ))

for name in sorted(names):
    print(f"hello, {name}")
```

Notice that once this program is executed, all information is lost. File I/O allows your program to store this information such that it can be used later.

- You can learn more in Python's documentation of sorted (https://docs.python.org/3/library/functions.html#sorted).

open

- `open` is a functionality built into Python that allows you to open a file and utilize it in your program. The `open` function allows you to open a file such that you can read from it or write to it.
- To show you how to enable file I/O in your program, let's rewind a bit and code as follows:

```
name = input("What's your name? ")

file = open("names.txt", "w")
file.write(name)
file.close()
```

Notice that the `open` function opens a file called `names.txt` with writing enabled, as signified by the `w`. The code above assigns that opened file to a variable called `file`. The line `file.write(name)` writes the name to the text file. The line after that closes the file.

- Testing out your code by typing `python names.py`, you can input a name and it saves to the text file. However, if you run your program multiple times using different names, you will notice that this program will entirely rewrite the `names.txt` file each time.
- Ideally, we want to be able to append each of our names to the file. Remove the existing text file by typing `rm names.txt` in the terminal window. Then, modify your code as follows:

```
name = input("What's your name? ")

file = open("names.txt", "a")
file.write(name)
file.close()
```

Notice that the only change to our code is that the `w` has been changed to `a` for "append". Rerunning this program multiple times, you will notice that names will be added to the file. However, you will notice a new problem!

- Examining your text file after running your program multiple times, you'll notice that the names are running together. The names are being appended without any gaps between each of the names. You can fix this issue. Again, remove the existing text file by typing `rm names.txt` in the terminal window. Then, modify your code as follows:

```
name = input("What's your name? ")

file = open("names.txt", "a")
file.write(f"{name}\n")
file.close()
```

Notice that the line with `file.write` has been modified to add a line break at the end of each name.

- This code is working quite well. However, there are ways to improve this program. It so happens that it's quite easy to forget to close the file.
- You can learn more in Python's documentation of open (https://docs.python.org/3/library/functions.html#open).

`with`

- The keyword `with` allows you to automate the closing of a file.
- Modify your code as follows:

```
name = input("What's your name? ")

with open("names.txt", "a") as file:
    file.write(f"{name}\n")
```

Notice that the line below `with` is indented.

- Up until this point, we have been exclusively writing to a file. What if we want to read from a file. To enable this functionality, modify your code as follows:

```
with open("names.txt", "r") as file:
    lines = file.readlines()

for line in lines:
    print("hello,", line)
```

Notice that `readlines` has a special ability to read all the lines of a file and store them in a file called lines. Running your program, you will notice that the output is quite ugly. There seem to be multiple line breaks where there should be only one.

- There are many approaches to fix this issue. However, here is a simple way to fix this error in our code:

```
with open("names.txt", "r") as file:
    lines = file.readlines()

for line in lines:
    print("hello,", line.rstrip())
```

Notice that `rstrip` has the effect of removing the extraneous line break at the end of each line.

- Still, this code could be simplified even further:

```
with open("names.txt", "r") as file:
    for line in file:
        print("hello,", line.rstrip())
```

Notice that running this code, it is correct. However, notice that we are not sorting the names.

- This code could be further improved to allow for the sorting of the names:

```
names = []

with open("names.txt") as file:
    for line in file:
        names.append(line.rstrip())

for name in sorted(names):
    print(f"hello, {name}")
```

Notice that `names` is a blank list where we can collect the names. Each name is appended to the `names` list in memory. Then, each name in the sorted list in memory is printed. Running your code, you will see that the names are now properly sorted.

- What if we wanted the ability to store more than just the names of students? What if we wanted to store both the student's name and their house as well?

CSV

- CSV stands for "comma separated values".
- In your terminal window, type `code students.csv`. Ensure your new CSV file looks like the following:

```
Hermoine,Gryffindor
Harry,Gryffindor
Ron,Gryffindor
Draco,Slytherin
```

- Let's create a new program by typing `code students.py` and code as follows:

```
with open("students.csv") as file:
    for line in file:
        row = line.rstrip().split(",")
        print(f"{row[0]} is in {row[1]}")
```

  Notice that `rstrip` removes the end of each line in our CSV file. `split` tells the compiler where to find the end of each of our values in our CSV file. `row[0]` is the first element in each line of our CSV file. `row[1]` is the second element in each line in our CSV file.

- The above code is effective at dividing each line or "record" of our CSV file. However, it's a bit cryptic to look at if you are unfamiliar with this type of syntax. Python has built-in ability that could further simplify this code. Modify your code as follows:

```
with open("students.csv") as file:
    for line in file:
        name, house = line.rstrip().split(",")
        print(f"{name} is in {house}")
```

  Notice that the `split` function actually returns two values: The one before the comma and the one after the comma. Accordingly, we can rely upon that functionality to assign two variables at once instead of one!

- Imagine that we would again like to provide this list as sorted output? You can modify your code as follows:

```
students = []

with open("students.csv") as file:
    for line in file:
        name, house = line.rstrip().split(",")
        students.append(f"{name} is in {house}")

for student in sorted(students):
    print(student)
```

  Notice that we create a `list` called `students`. We `append` each string to this list. Then, we output a sorted version of our list.

- Recall that Python allows for `dictionaries` where a key can be associated with a value. This code could be further improved

```
students = []

with open("students.csv") as file:
    for line in file:
        name, house = line.rstrip().split(",")
        student = {}
        student["name"] = name
        student["house"] = house
        students.append(student)

for student in students:
    print(f"{student['name']} is in {student['house']}")
```

Notice that we create an empty dictionary called `student`. We add the values for each student, including their name and house into the `student` dictionary. Then, we append that student to the `list` called `students`.

- We can improve our code to illustrate this as follows:

```
students = []

with open("students.csv") as file:
    for line in file:
        name, house = line.rstrip().split(",")
        student = {"name": name, "house": house}
        students.append(student)

for student in students:
    print(f"{student['name']} is in {student['house']}")
```

Notice that this produces the desired outcome, minus the sorting of students.

- Unfortunately, we cannot sort the students as we had prior because each student is now a dictionary inside of a list. It would be helpful if Python could sort the `students` list of `student` dictionaries that sorts this list of dictionaries by the student's name.

- To implement this in our code, make the following changes:

```
students = []

with open("students.csv") as file:
    for line in file:
        name, house = line.rstrip().split(",")
        students.append({"name": name, "house": house})


def get_name(student):
    return student["name"]


for student in sorted(students, key=get_name):
    print(f"{student['name']} is in {student['house']}")
```

Notice that `sorted` needs to know how to get the key of each student. Python allows for a parameter called `key` where we can define on what "key" the list of students will be sorted. Therefore, the `get_name` function simply returns the key of `student["name"]`. Running this program, you will now see that the list is now sorted by name.

- Still, our code can be further improved upon. It just so happens that if you are only going to use a function like `get_name` once, you can simplify your code in the manner presented below. Modify your code as follows:

```
students = []

with open("students.csv") as file:
    for line in file:
        name, house = line.rstrip().split(",")
        students.append({"name": name, "house": house})

for student in sorted(students, key=lambda student: student["name"]):
    print(f"{student['name']} is in {student['house']}")
```

Notice how we use a `lambda` function, an anonymous function, that says "Hey Python, here is a function that has no name: Given a `student`, access their `name` and return that to the `key`.

- Unfortunately, our code is a bit fragile. Suppose that we changed our CSV file such that we indicated where each student grew up. What would be the impact of this upon our program? First, modify your `students.csv` file as follows:

```
Harry,"Number Four, Privet Drive"
Ron,The Burrow
Draco,Malfoy Manor
```

Notice how running our program how will produce a number of errors.

- Now that we're dealing with homes instead of houses, modify your code as follows:

```
students = []

with open("students.csv") as file:
    for line in file:
        name, home = line.rstrip().split(",")
        students.append({"name": name, "home": home})

for student in sorted(students, key=lambda student: student["name"]):
    print(f"{student['name']} is in {student['home']}")
```

Notice that running our program still does not work properly. Can you guess why?

- The `ValueError: too many values to unpack` error produced by the compiler is a result of the fact that we previously created this program expecting the CSV file is `split` using a `,` (comma). We could spend more time addressing this, but indeed someone else has already developed a way to "parse" (that is, to read) CSV files!

- Python's built-in `csv` library comes with an object called a `reader`. As the name suggests, we can use a `reader` to read our CSV file despite the extra comma in "Number Four, Privet Drive". A `reader` works in a `for` loop, where each iteration the `reader` gives us another row from our CSV file. This row itself is a list, where each value in the list corresponds to an element in that row. `row[0]`, for example, is the first element of the given row, while `row[1]` is the second element.

```
import csv

students = []

with open("students.csv") as file:
    reader = csv.reader(file)
    for row in reader:
        students.append({"name": row[0], "home": row[1]})

for student in sorted(students, key=lambda student: student["name"]):
    print(f"{student['name']} is from {student['home']}")
```

  Notice that our program now works as expected.

- Up until this point, we have been relying upon our program to specifically decide what parts of our CSV file are the names and what parts are the homes. It's better design, though, to bake this directly into our CSV file by editing it as follows:

```
name,home
Harry,"Number Four, Privet Drive"
Ron,The Burrow
Draco,Malfoy Manor
```

  Notice how we are explicitly saying in our CSV file that anything reading it should expect there to be a name value and a home value in each line.

- We can modify our code to use a part of the `csv` library called a `DictReader` to treat our CSV file with even more flexibilty:

```
import csv

students = []

with open("students.csv") as file:
    reader = csv.DictReader(file)
    for row in reader:
        students.append({"name": row["name"], "home": row["home"]})

for student in sorted(students, key=lambda student: student["name"]):
    print(f"{student['name']} is in {student['home']}")
```

  Notice that we have replaced `reader` with `DictReader`, which returns one dictionary at a time. Also, notice that the compiler will directly access the `row` dictionary, getting the `name` and `home` of each student. This is an example of coding defensively. As long as the person designing the CSV file has inputted the correct header information on the first line, we can access that information using our program.

- Up until this point, we have been reading CSV files. What if we want to write to a CSV file?
- To begin, let's clean up our files a bit. First, delete the `students.csv` file by typing `rm students.csv` in the terminal window. This command will only work if you're in the same folder as your `students.csv` file.
- Then, in `students.py`, modify your code as follows:

```
import csv

name = input("What's your name? ")
home = input("Where's your home? ")

with open("students.csv", "a") as file:
    writer = csv.DictWriter(file, fieldnames=["name", "home"])
    writer.writerow({"name": name, "home": home})
```

  Notice how we are leveraging the built-in functionality of `DictWriter`, which takes two parameters: the `file` being written to and the `fieldnames` to write. Further, notice how the `writerow` function takes a dictionary as its parameter. Quite literally, we are telling the compiler to write a row with two fields called `name` and `home`.
- Note that there are many types of files that you can read from and write to.
- You can learn more in Python's documentation of CSV (https://docs.python.org/3/library/csv.html).

Binary Files and `PIL`

- One more type of file that we will discuss today is a binary file. A binary file is simply a collection of ones and zeros. This type of file can store anything including, music and image data.
- There is a popular Python library called `PIL` that works well with image files.
- Animated GIFs are a popular type of image file that has many image files within it that are played in sequence over and over again, creating a simplistic animation or video effect.
- Imagine that we have a series of costumes, as illustrated below.
- Here is `costume1.gif`.

- Here is another one called `costume2.gif`. Notice how the leg positions are slightly different.

- Before proceeding, please make sure that you have downloaded the source code files from the course website. It will not be possible for you to code the following without having the two images above in your possession and stored in your IDE.
- In the terminal window type `code costumes.py` and code as follows:

```
import sys

from PIL import Image

images = []

for arg in sys.argv[1:]:
    image = Image.open(arg)
    images.append(image)

images[0].save(
    "costumes.gif", save_all=True, append_images=[images[1]], duration=200, lo
)
```

  Notice that we import the `Image` functionality from `PIL`. Notice that the first `for` loop simply loops through the images provided as command-line arguments and stores theme into the `list` called `images`. The `1:` starts slicing `argv` at its second element. The last lines of code saves the first image and also appends a second image to it as well, creating an animated gif. Typing `python costumes.py costume1.gif costume2.gif` into the terminal. Now, type `code costumes.gif` into the terminal window, and you can now see an animated GIF.
- You can learn more in Pillow's documentation of PIL (https://pillow.readthedocs.io/).

Summing Up

Now, we have not only seen that we can write and read files textually—we can also read and write files using ones and zeros. We can't wait to see what you achieve with these new abilities

next.

- File I/O
- `open`
- `with`
- CSV
- `PIL`