String Formats
File Input/Output

CS 8: Introduction to Computer Science, Winter 2019
Lecture #11

Ziad Matni, Ph.D.
Dept. of Computer Science, UCSB

Administrative

- **Hw06 – due next week Monday (2/25)**
- Lab time this week is for **Project**
  - Due in the last week of class
- **Lab06** will be issued next week

- **MIDTERM #2 is COMING NEXT WEEK!!!**
  - On Wednesday Feb. 27th
  - Practice exam will be on website soon

Midterm #2

- **What's going to be on it?**
  - Functions
  - Conditionals
  - Loops
  - String Formats
  - File I/O
  - Random Numbers (and other Math stuff)*

* depending on how far we get

Lecture Outline

- Using the **format()** function
- File Input / Output

Formatted Outputs

- You know these already:

```
print(42)           # prints 42 and then a newline (wow)
print(42, "!")      # prints '42 !' and then a newline (note the space)
print(42, end="")   # prints 42 WITHOUT a newline character
```

- Expanding on the above…

```
print(42, end="!")  # prints 42! WITHOUT a newline character (note NO space!)
```

Using the **.format()** Function with Strings

- You can print an output while you ***define*** your general format!

Example:

```
hour = 12
minute = 55
second = 31
```

Note: the {0} refers to hour (the 0th argument), the {1} to minute (the 1st argument), etc…
THIS ORDER MATTERS!!

Example, what would happen if I switched {0} and {1} in here?

If you do this: `'{0}:{1}:{2}'.format(hour, minute, second)`
You get this: `12:55:31` *(it's a string output)*

Let's try it out!

More on .format()

- You can define how many spaces an object occupies when printed

Example:

```
>>> a = 19
>>> b = 42
>>> '{0:3}***{1:5}'.format(a, b)
' 19***   42'
```

*Refers to the 0th item (that is, variable **a**)*

Refers to the total number of spaces you want to format

3 spaces

5 spaces

Let's try it out!

YET MORE on **.format()**

- With strings instead of numbers

Example:

```
>>> a = "Be"
>>> b = "Mine!"
>>> '{:7}{:>7}'.format(a, b)
'Be       Mine!'
```

7 spaces *7 spaces*

*Save 7 spaces for var. **a** and **left justify a***
Put any extra spaces AFTER it

*Save 7 spaces for var. **b** and **right justify b***
Put any extra spaces BEFORE it

What happens if you run out of space?
Does it:

***a.** cut out the string to make it fit?*
***b.** still print out the string even if it's longer than the space format?*

Let's try it out!

.format() with Floating Points

- If you say, print(100/3), you get: 33.333333333333336
- What if you wanted to instill some precision on your decimal values?

Example:

```
>>> '{:7.3f}'.format(100/3)
' 33.333'
```

7 spaces

Save 7 spaces for the floating point.
Put 3 numbers after the decimal point

Let's try it out!

.format() with Floating Points using Engineering Notation

- If you say, print(100/3), you get: 33.333333333333336

Example:

```
>>> '{:10.1e}'.format(100/3)
'   3.3e+01'
```

10 spaces

Save 10 spaces for the floating point and use engineering notation.

Let's try it out!

More Examples

- Go to your textbook and read through all the examples in **Ch. 4.2**

- There are other types of format
- **CHECK THOSE OUT TOO!!!**

Files

- Mostly handled like any *sequential data type*
  - What's an example of a data type that can be read sequentially?
- Files are **a sequence of characters** *if they are text files*, or **a sequence of bits** *if they are binary file*
  - What are bits??
- Can you name some common file *types* that are textual? Or that are binary?

Why Use Files?

4 Good Reasons:

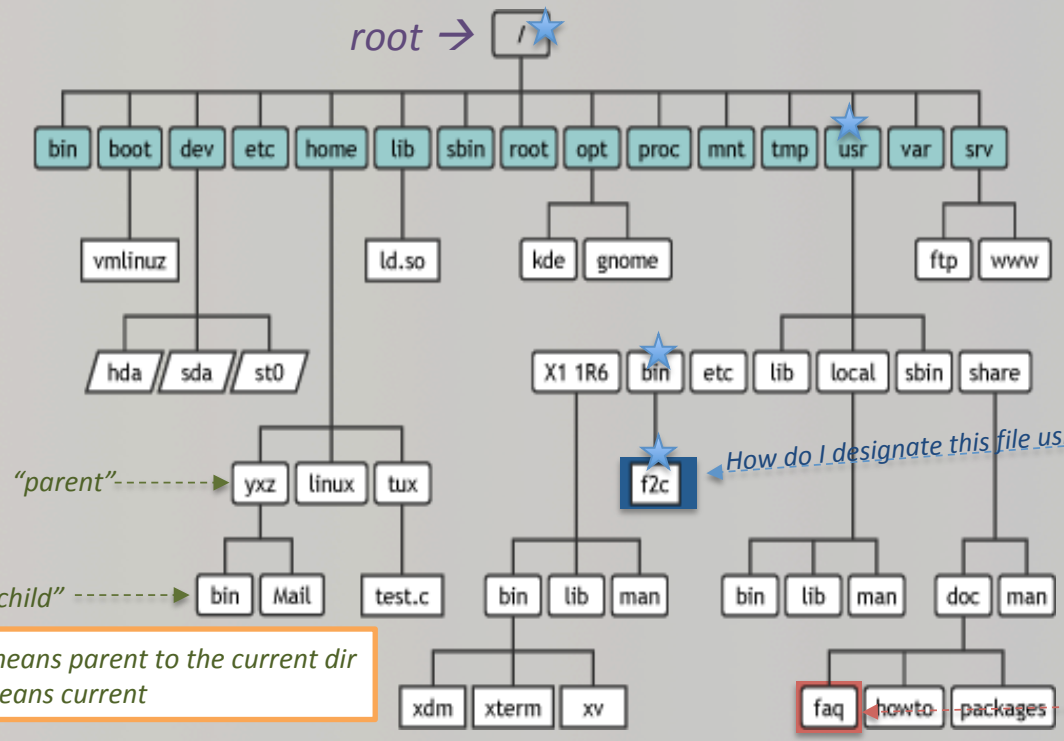
- Files allow you to store data **permanently** and **conveniently**!
- Data output that goes to a file stays there **after the program ends**
  - You can usually view the data without the need of a Python program
- An input data file **can be used over and over again**
  - No need to type data again and again for testing
- Files allow you to deal with larger data sets
  - Imagine putting all historical weather data for the USA in one list or string!!! 😐

Input and Output in Computers

- Input and output (or **I/O**) are 2 of the main components of any computer

- There are different types of I/O
  - What we call "**standard output**" is usually the screen
  - What we call "**standard input**" is usually from the keyboard
  - But there ARE other ways to get I/O
    - Like using files to write to (output) or to read from (input)

Organization of Files in a Computer



Is done **hierarchically**
Uses **folders** (aka directories)
Starts at the "**root**" directory
designated with a /

File I/O: Simple Example

Example of READING from a file

```
infile = open('DataFile.txt', 'r')

line = infile.read()
# read everything in one string!

print(line)

infile.close()
# DON'T FORGET TO CLOSE!!!
```

Example of WRITING to a file

```
outfile = open('MyOuts.txt', 'w')

x = 3
y = 4
n = (x + y)**y

outfile.write('Number', n)

outfile.close()
# DON'T FORGET TO CLOSE!!!
```

Different Ways of Reading File Input

```
line = infile.read()
                        # Read everything into 1 string
line = infile.read(n)
                        # Read the first n chars into 1 string
line = infile.readline()
                        # Read 1 line (ends in '\n') into 1 string
line = infile.readlines()
                        # Read all lines into 1 list
```

DEMO!
Let's try it!

YOUR TO-DOs

- **HW6** (due on **Monday, 2/25**)

- Work on your Project Assignment!

</LECTURE>