Lecture 20

Chapter 8 Input, Output, and Files

8.1 Standard Input and Output (review)

8.2 Formatted Strings

8.3 Working with Files

8.5 Case Studies

8.1 Standard Input and Output

The simplest form of receiving input from a user is through the use of the `input` function.

Syntax: `input(['prompt'])`

When called, the system waits while the user types in a string of characters. If Enter key is pressed, the function returns the string of entered characters, *up to but not including the final newline*.

(The function reads a line from input, converts it to a string (stripping a trailing newline), and returns that.)

8.1 Standard Input and Output

The simplest form of receiving input from a user is through the use of the `input` function.

Syntax: `input(['prompt'])`

When called, the system waits while the user types in a string of characters. If Enter key is pressed, the function returns the string of entered characters, *up to but not including the final newline*.

(The function reads a line from input, converts it to a string (stripping a trailing newline), and returns that.)

This function corresponds to `raw_input` function in Python 2
Syntax: `raw_input(['prompt'])`

After a string is entered, we can use `eval` method, if the input is expected to be a number or a formula and all the variables in it are pre-defined in our program.

8.1 Standard Input and Output

`print` method : simplest form for generating output

- accepts 0 or more subsequent arguments separated by commas.

Note: prior to Python 3.0 print was not formally a function, but instead a keyword of the language. Therefore its arguments were not specified within parentheses.

8.1 Standard Input and Output

`print` method : simplest form for generating output

- accepts 0 or more subsequent arguments separated by commas.

Note: prior to Python 3.0 print was not formally a function, but instead a keyword of the language. Therefore its arguments were not specified within parentheses.

Several subtleties in the behavior of the command:

1. the arguments may be literals, identifiers, or compound expressions;
2. if an individual argument is not already an instance of the string class, it is automatically converted to a string;
3. when multiple arguments are given, an explicit space is automatically inserted between successive arguments.
4. by default, the `print` command generates one final newline character after printing all the arguments (which can be suppressed, see next slide)

8.1 Standard Input and Output

`print` method

1. we can ***break long statements into multiple lines***:

```
print("Hello, today's temperature is", degrees, \
        "degrees, but it feels like", feels, \
        "degrees.")
```

same will be for almost any statement, for example:

```
result = 1*2 + 1*2**2 + 1 * 3**2 + 4 * 2**5 + \
            6 * 2**6 + 7 * 2**7
```

8.1 Standard Input and Output

`print` method

1. we can ***break long statements into multiple lines***:

```
print(“Hello, today's temperature is”, degrees, \
        “degrees, but it feels like”, feels, \
        “degrees.”)
```

same will be for almost any statement, for example:

```
result = 1*2 + 1*2**2 + 1 * 3**2 + 4 * 2**5 + \
          6 * 2**6 + 7 * 2**7
```

2. The ***newline*** character

```
print(“One”)
print(“Two”)
print(“Three”)
```

```
print(“One”, end=' ')
print(“Two”, end = ' ')
print(“Three”, end = ' ')
```

- no space between the single quotes, or anything else

8.1 Standard Input and Output

`print` method

3. we can specify an item separator: ***break long statements into multiple lines***:

```
print(“Hello”,”How”,”are”,”you”,today?”,sep = “***”)
```

result: Hello***How***are***you***today?

8.1 Standard Input and Output

print method

3. we can specify an item separator: ***break long statements into multiple lines***:

print(“Hello”,”How”,”are”,”you”,today?”,sep = “***”)

result: Hello***How***are***you***today?

4. we can use escape characters:
\n new line
\t tabulation (skips few spaces)
\' single quote will be printed
\” double quote will be printed
\\ backslash character will be printed

example: print(“One \t two \t \”three\””)

result: One two "three"

8.2 Formatted Strings

In ***Python 2*** to output formatted strings we can use the % operator (modulo).

% sign starts the conversion specifier.

8.2 Formatted Strings

In ***Python 3*** to output formatted strings we can use built-in format function. It takes two arguments: a numeric value, and a format specifier

8.2 Formatted Strings

Examples:

```
num,denom=3.123,4.234
print(format(num,'.2f'),"/",format(denom,'.2f'),\
        "=",format(num/denom,'.2f'))
```

type designator (conversion type)

number of decimal places to display (*precision*)

Produces: 3.12 / 4.23 = 0.74

8.2 Formatted Strings

Examples:

```
num,denom=3.123,4.234
print(format(num,'.2f'),"/",format(denom,'.2f'),\
      "=",format(num/denom,'.2f'))
```

type designator (conversion type)

number of decimal places to display (*precision*)

Produces: 3.12 / 4.23 = 0.74

```
n = 23
print(format(n,'4d'))
```

minimum field width

Produces: _ _23

Underscores stand for two spaces that are added in front of 23.

8.2 Formatted Strings

Examples:

```
num,denom=3.123,4.234
print(format(num,'.2f'),"/",format(denom,'.2f'),\
      "=",format(num/denom,'.2f'))
```

type designator (conversion type)

number of decimal places to display (*precision*)

Produces: 3.12 / 4.23 = 0.74

minimum field width

```
n = 23
print(format(n,'4d'))
```

Produces: _ _23

Underscores stand for two spaces that are added in front of 23.

```
n = 23
print(format(n,'04d'))
```

Produces: 0023

Number is padded with two leading zeros

8.2 Formatted Strings

6. **Conversion type**.

Conversion	Meaning
'd'	Signed integer decimal.
'i'	Signed integer decimal.
'o'	Signed octal value.
'x'	Signed hexadecimal (lowercase).
'X'	Signed hexadecimal (uppercase).
'e'	Floating point exponential format (lowercase).
'E'	Floating point exponential format (uppercase).
'f' or 'F'	Floating point decimal format.
'c'	Single character (accepts integer or single character string).
's'	String (converts any Python object using str()).

8.2 Formatted Strings

Example:

```
data = [1024,4,16,32]
print(format(data[0],'4d'),format(data[1],'4d'),\
      format(data[2],'4d'),format(data[3],'4d'))
print('---- ' * len(data))
```

produces:
```
1024    4   16   32
---- ---- ---- ----
```

you can find more about string formatting in Python 3
(Google for it, or check the reference manual)

All the examples are gathered in program experiments.py

8.3 Working with Files

Programs must be able to read data from file and to write data to files. It is especially needed when we have a large volume of data.
Python supports a built-in class `file` to manipulate files on the computer.

Constructor of Python's file class accepts two parameters:
- *file name* (as string), and
- *access mode* (as string, optional)
  - r – for reading (default mode)
  - w – for (over)writing
  - a – for appending to the end of the file

8.3 Working with Files

Programs must be able to read data from file and to write data to files. It is especially needed when we have a large volume of data.
Python supports a built-in class `file` to manipulate files on the computer.

Constructor of Python's file class accepts two parameters:
- *file name* (as string), and
- *access mode* (as string, optional)
  - r – for reading (default mode)
  - w – for (over)writing
  - a – for appending to the end of the file

Example:

```
file1 = open('inputData.txt')
```

- file inputData will be open in read-only mode

```
file2 = open('outputData.txt','w')
```

- file outputData will be open for writing (re-writing)

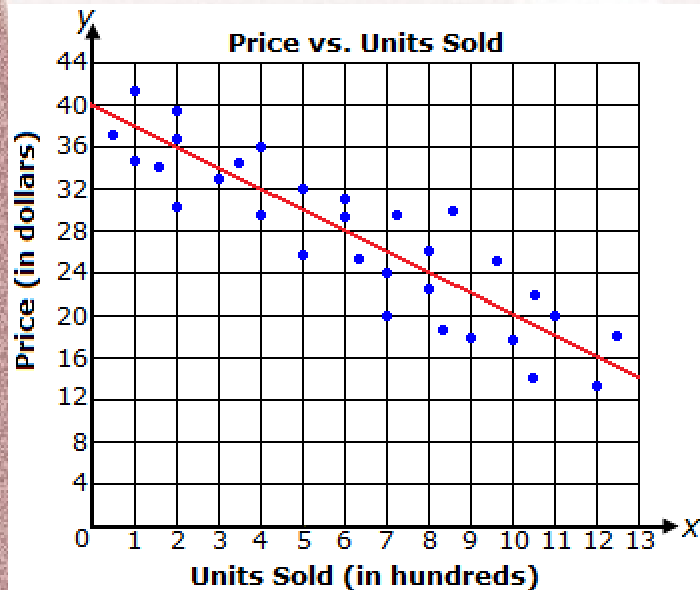
Selected behaviors of Python's file class:

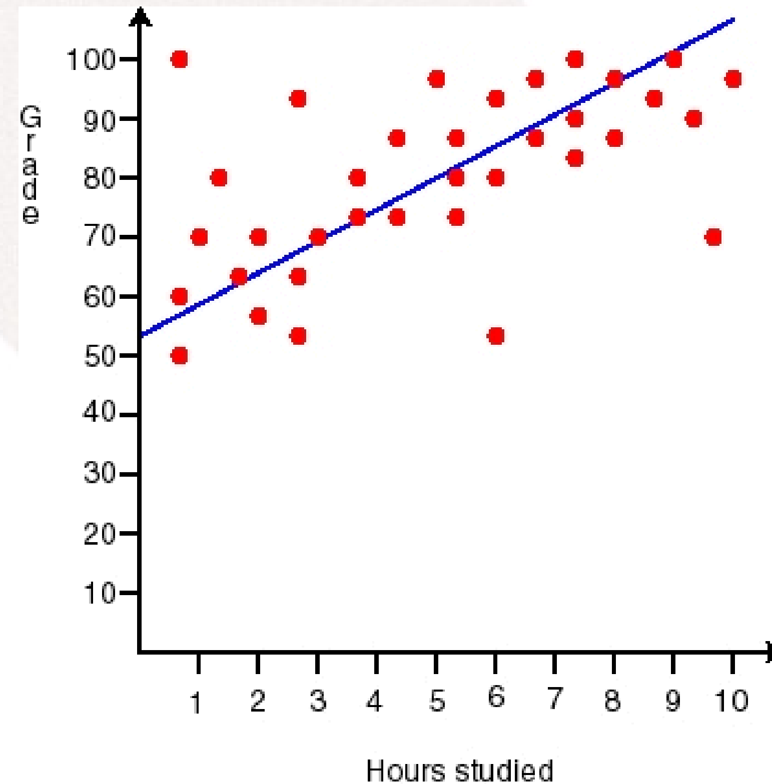
Syntax	Semantics
open()	Returns a file object (used with two arguments)
close()	Disconnects the file object from the associated file (saving it, if necessary)
flush()	Flushes the buffer of written characters, saving the underlying file
read()	Returns a string representing the (remaining) contents of the file
read(size)	Returns a string representing the specified number of bytes next in the file
readline()	Returns a string representing the next line of the file
readlines()	Returns a list of strings representing the remaining lines of the file
write(s)	Writes the given string to the file. No newlines are added.
writelines(seq)	Writes each of the strings to the file. No newlines are added.
for line in f	Iterates through the file~f, one line at a time

Example: Statistics Project

Example: on pages 295-296 we have a project **8.18-8.19**.

Write a program that reads a data set of two-dimensional points and calculates the "*line of best fit*" (or *regression line*) for that point set (namely, the line that minimizes the sum of the squares of the vertical distances between the points and the line)





Example: Statistics Project

Example: on pages 295-296 we have a project **8.18-8.19**.

We may assume that the file is formatted so that each line describes a single point, denoted by x and y coordinate values separated by a space.

The equation of the line is in the form $y=mx+b$,
m is the slope of the line,
b is the y-intercept.
m and b are computed using the formulas (next slide)

We will do only tiny part of that project:
get data from the file and find find two sums.

Example: Statistics Project

Example of input:
Content of the file *data.txt*:

12 23
14 45
-4 -20
-15 -60
15 40
17 60
-10 -48

That's what we will do:

	x	y
	12	23
	14	45
	-4	-20
	-15	-60
	15	40
	17	60
	-10	-48
sum	29	40

Example: Statistics Project

a sketch of our program:

```
def main():
  data = getData() # get data from the input file

  display(data) # display data on the screen

  x,y = findSums(data) # find sums

main()
```

Example: Statistics Project

First, given the data file we'd like to be able to compile the following list:

data = [(12,23),(14,45),(-4,-20),(-15,-60),(15,40),(17,60),(-10,-48)]

Then we will do the following:

```
n = len(data)

    sumx, sumy = 0,0

    for i in range(n):
        sumx += data[i][0]
        sumy += data[i][1]
```

Did you notice that we have “two-dimensions”?

1st	2nd columns	
(12,	23)	1st
(14,	45)	2nd
(-4,	-20)	3rd
(-15,	-60)	4th
(15,	40)	5th
(17,	60)	6th
(-10,	-48)	7th rows

Example: Statistics Project

See the program project18.py

Homework Assignment

p. 293-294 / 8.6 and 8.11

Addition for 8.6: if while searching through the file you find that a year is negative, raise a Value Exception, saying that a birth year cannot be negative.

For both of the programs: catch IOError and deal with them as it was dealt in project 18.