Python Programming

Lecture 3

2/15/10

Today

- Quick Review
- Input and Output mechanism
  - String Formatting
- File Handling
- Command Line Arguments

Input and Output

- What I/O have you seen already?
- How does I/O work on a computer?
- What can you accomplish with I/O?

`print` function

- General Output function, sends to **`stdout`**
- No need for parenthesis (in v. 2.*)
- Inserts a new line, but *not* when there is a trailing comma
- Arguments are comma separated
    - Deliminator (space) inserted between arguments
- Automatically casts to a string, using **`str()`** **`repr()`**, before printing
- Demonstration!

`print` example

```
>>> for i in
range(10): print i
...
0
1
2
3
4
5
6
7
8
9
>>> for i in
range(10): print i,
...
0 1 2 3 4 5 6 7 8 9
```

```
>>>for i in range(10):
...     print "%s" %(str(i)*10), i,
...
0000000000 0 1111111111 1 2222222222 2
3333333333 3 4444444444 4 5555555555 5
6666666666 6 7777777777 7 8888888888 8
9999999999 9
```

String Formatting (side note)

```
>>> '%d: %s, %f' %(1,'spam',1.2)
'1: spam, 1.200000'
>>> '%d: %s, %f' %('spam',1,1.2)
TypeError, int argument required
```

- The format is a place holder to be replaced *in order* with the tuple following the '%'
    - %d ~ integers
    - %s ~ strings
    - %f ~ floats
- what if you want '%'
- Other formats, '\n' , '\t',

We have output, what about input?

- No `fgets()` nightmares
- No `System.in.readline()` headaches
- Python makes it easy.
  - It's built in
  - As expected.

Input

- `raw_input()`
  - Read's a string from standard input. The trailing newline is stripped, and returned.
- `EOFError`
  - End of File, CTL-D, CTL-Z<enter> (windows)
- `try except` semantics
  - Just like try/catch in java
  - Signal Handling in C

Simple introduction to exception

- This is an event that can modify the control flow of a program
- Remember `KeyError`, `IndexError` or `TypeError`
  - these are extension of a base exception
- An exception is an event
  - Something Good/Bad/Blah Happened!
  - You have to handle it, or the program crashes

Very simple `try except`

- What is a try except coding?
  - Try this code block, if something happens, handle it
  - We are *trying* to read from **`stdin`**,
  - It takes *exception* when reading End Of File
- We will cover exceptions in more detail later
- Sometimes, it is ok to have an un-handled exception, choose when it is appropriate and when it is not (**mostly not though**)
- Let's see what this looks like!! Demonstration!

Echo Program

```
#!/usr/bin/python

while True:
    try:
        input = raw_input(">")
        print "You said:",input
    except EOFError:
        print "good bye"
        break
```

```
$ ./echo.py
>hello
You said: hello
>what
You said: what
>good bye
$
```

In and Out

- We have output to **`stdout`** using **`print`**
- We can get input from **`stdin`** using **`raw_input()`**
- What other I/O are we missing?

Files

- What is a file?
- How are they accessed?
- Are all files equal?
- How do we create new ones?

The `file` type

- The **`file`** type is a basic type
- It stores file descriptor/stream information
- **`fd = open(file_name, mode)`**
  - Has a default mode, what is it?
- **`fd.close()`**
  - Closes a file type
  - should always close open file descriptors
- Try out **`dir(file)`** or **`help(file)`** for more info

modes

- "r" read
- "w" write
- "a" append
- "+" for simultaneous writing/reading
- help(file) for more info
- What is **`open(f,"wa")`**?
  - Open to write and all writes happen at the end of the file

The `file` type (cont)

- Just like other basic types, we have a slew of access functions
- **`data = fd.read([size])`**
  - Read up to **`size`**, or until **`EOF`** is reached
- **`fd.write(data)`**
  - Write the string **`data`** to the file
- Other very useful functions
  - **`readline(),readlines(),writelines()`**

`print` and `>>`

- The **`write()`** function is less powerful then the **`print`** function
- Well, use the print function
  - **`print >> myfd,'SpamEnEggs'*2`**
- The **`>>`** operator tells **`print`** to use some other file descriptor rather then **`stdout`**
- What if we want this for all **`print`**'s
  - reassign **`stdout`** (shown later)

File Copy Program

```
#!/usr/bin/python

f_in = "input.txt"
f_out = f_in+".cpy"

fd = open(f_in,"r")
file = fd.read()
fd.close()

fd = open(f_out,"w")
fd.write(file)
fd.close()
```

Another File Copy Program

```
#!/usr/bin/python

f_in = "input.txt"
f_out = f_in+".cpy"
fd_in = open(f_in,"r")
fd_out = open(f_out,"w")

fd_out.writelines(fd_in.readlines())

fd_in.close()
fd_out.close()
```

Just One More I/O

- What about command line arguments?
  - How does this work in C and Java?
- When you execute a program in python ...
  - What is really running?
  - Who gets the arguments?

The `sys` Module

- You will use the **`sys`** module often
    - *This module provides access to some objects used or maintained by the interpreter and to functions that interact strongly with the interpreter*
- **`stdin, stdout, argv, exit()`**
- **`ps1, ps2`**
- Very useful stuff
    - **`traceback`** information,

sys and Command Line Args

- **sys.argv**
  - It's a list of the command line args to the program
  - **sys.argv[0]** is the name of the program by convention

```
#!/usr/bin/python
# argPrinter2.py
import sys

if __name__ == "__main__":
    for arg in sys.argv:
        if arg == “Shazbot”:
            print “NanoNano”
        else:
            print arg
```

Some Other Things in `sys`

```
help(sys)
...
    argv = ['']
    builtin_module_names = ('__builtin__', '__main__', '_ast', '_codecs', ...
    byteorder = 'little'
    copyright = 'Copyright (c) 2001-2008 Python Software Foundati...ematis...
    exc_value = TypeError('arg is a built-in module',)
    exec_prefix = '/usr'
    executable = '/usr/bin/python'
    hexversion = 33882864
    last_value = AttributeError("'module' object has no attribute 'module'...
    maxint = 2147483647
    maxunicode = 1114111
    ...
    platform = 'linux2'
    prefix = '/usr'
    ps1 = '>>> '
    ps2 = '... '
    stderr = <open file '<stderr>', mode 'w' at 0xb7f890b0>
    stdin = <open file '<stdin>', mode 'r' at 0xb7f89020>
    stdout = <open file '<stdout>', mode 'w' at 0xb7f89068>
    subversion = ('CPython', 'tags/r252', '60911')
    version = '2.5.2 (r252:60911, Sep 30 2008, 15:41:38) \n[GCC 4.3.2 2008...
    version_info = (2, 5, 2, 'final', 0)
    warnoptions = []
```

More I/O with `sys` module

- We can access the files **`stdin`** and **`stdout`** from **`sys`**
  - **`sys.stdin, sys.stdout, sys.stderr`**
- Can read and write from these files
  - **`sys.stdin.read()`**
  - **`sys.stdout.write(),sys.stderr.write()`**
- What happens with **`print, raw_input()`** if we do this?
  - **`sys.stdin = open("input_file")`**
  - **`sys.stdout = open("output_file")`**

Questions?

- Homework?
- OOP: Class vs. Instance?
- Next Week
    - Modules
    - Packages
    - Scoping and Nesting