Introduction to Python
Exceptions, Unicode, Text Processing

Christopher Barker

UW Continuing Education / Isilon

July 11, 2012

Table of Contents

Last Class

Lab from end of last class?

LAB

`def count_them(letter):`

- prompts the user to input a letter
- counts the number of times the given letter is input
- prompts the user for another letter
- continues until the user inputs "x"
- returns the count of the letter input

`def count_letter_in_string(string, letter):`

- counts the number of instances of the letter in the string
- ends when a period is encountered
- if no period is encountered – prints "hey, there was no period!"

Questions?

Any Questions about:

- Last class ?
- Reading ?
- Homework ?

Homework review

Homework notes

subprocesses

Subprocesses

```
#easy:
os.popen('ls').read()

#even easier:
os.system('ls')

# but for anything more complicated:
pipe = \
  subprocess.Popen("ls", stdout=subprocess.PIPE).stdout
```

reload

module importing and reloading

```
In [190]: import module_reload

In [191]: module_reload.print_something()
I'm printing something

# change it...
In [196]: reload(module_reload)
Out[196]: <module 'module_reload' from 'module_reload.py'>

In [193]: module_reload.print_something()
I'm printing something else
```

Module Reloading

```
In [194]: from module_reload import this

# change it...

In [196]: reload(module_reload)
Out[196]: <module 'module_reload' from 'module_reload.py'>

In [197]: module_reload.this
Out[197]: 'this2'

In [198]: this
Out[198]: 'this'
```

repr vs. str

repr() vs str()

```
In [200]: s = "a string\nwith a newline"

In [203]: print str(s)
a string
with a newline

In [204]: print repr(s)
'a string\nwith a newline'
```

repr vs. str

```
eval(repr(something)) == something
```

```
In [205]: s2 = eval(repr(s))

In [206]: s2
Out[206]: 'a string\nwith a newline'
```

Strings

A string literal creates a string type

```
"this is a string"
```

Can also use `str()`

```
In [256]: str(34)
Out[256]: '34'
```

or "back ticks"

```
In [258]: `34`
Out[258]: '34'
```

(demo)

The String Type

Lots of nifty methods:

```
s.lower()
s.upper()
    ...
s.capitalize()
s.swapcase()
s.title()
```

http://docs.python.org/library/stdtypes.html#index-23

The String Type

Lots of nifty methods:

```
x in s
s.startswith(x)
s.endswith
...
s.index(x)
s.find(x)
s.rfind(x)
```

http://docs.python.org/library/stdtypes.html#index-23

The String Type

Lots of nifty methods:

```
s.split()
s.join(list)
...
s.splitlines()
```

`http://docs.python.org/library/stdtypes.html#index-23`

(demo – join)

The string module

Lots of handy constants, etc.

```
string.ascii_letters
string.ascii_lowercase
string.ascii_uppercase
string.letters
string.hexdigits
string.whitespace
string.printable
string.digits
string.punctuation
```

(and the string methods – legacy)

String Literals

Common Escape Sequences

```
\\  Backslash (\)
\a  ASCII Bell (BEL)
\b  ASCII Backspace (BS)
\n  ASCII Linefeed (LF)
\r  ASCII Carriage Return (CR)
\t  ASCII Horizontal Tab (TAB)
\ooo  Character with octal value ooo
\xhh  Character with hex value hh
```

(http://docs.python.org/release/2.5.2/ref/strings.html)

Raw Strings

Escape Sequences Ignored

```
In [408]: print "this\nthat"
this
that
In [409]: print r"this\nthat"
this\nthat
```

Gotcha:

```
In [415]: r"\"
SyntaxError: EOL while scanning string literal
```

(handy for regex, windows paths...)

Building Strings

Please don't do this:

```
'Hello ' + name + '!'
```

(much)

Building Strings

Do this instead:

```
'Hello %s!' % name
```

much faster and safer:

easier to modify as code gets complicated

`http://docs.python.org/library/stdtypes.html#string-formatting-operations`

Joining Strings

The Join Method:

```
In [289]: t = ("some", "words","to","join")

In [290]: " ".join(t)
Out[290]: 'some words to join'

In [291]: ",".join(t)
Out[291]: 'some,words,to,join'

In [292]: "".join(t)
Out[292]: 'somewordstojoin'
```

String Formatting

The string format operator: %

```
In [261]: "an integer is: %i"%34
Out[261]: 'an integer is: 34'

In [262]: "a floating point is: %f"%34.5
Out[262]: 'a floating point is: 34.500000'

In [263]: "a string is: %s"%"anything"
Out[263]: 'a string is: anything'
```

String Formatting

multiple arguments:

```
In [264]: "the number %s is %i"%('five', 5)
Out[264]: 'the number five is 5'

In [266]: "the first 5 numbers are: %i, %i, %i, %i, %i"%(1,
Out[266]: 'the first 5 numbers are: 1, 2, 3, 4, 5'
```

String formatting

Gotcha

```
In [127]: "this is a string with %i formatting item"%1
Out[127]: 'this is a string with 1 formatting item'

In [128]: "string with %i formatting %s: "%2, "items"
TypeError: not enough arguments for format string

# Done right:
In [131]: "string with %i formatting %s"%(2, "items")
Out[131]: 'string with 2 formatting items'

In [132]: "string with %i formatting item"%(1,)
Out[132]: 'string with 1 formatting item'
```

String formatting

Named arguments

```
'Hello %(name)s!'%{'name':'Joe'}
'Hello Joe!'

'Hello %(name)s, how are you, %(name)s!' %{'name':'Joe'}
'Hello Joe, how are you, Joe!'
```

That last bit is a dictionary (next week)

Advanced Formatting

The format method

```
In [283]: 'Hello {0}!'.format(name)
Out[283]: 'Hello Joe!'

In [284]: 'Hello {name}!'.format(**dictionary)
Out[284]: 'Hello Joe!'
```

pick one (probably string formatting):
– get comfy with it

LAB

Fun with strings

- Rewrite:
  `the first 3 numbers are: %i, %i, %i"%(1,2,3)`
  for an arbitrary number of numbers...
- Play with string formatting a bit more.
- Write a (really simple) mail merge program
- implement rot13 decoding

  decode this message:
  Zntargvp sebz bhgfvqr arne pbeare

LAB

ROT13 encryption

Applying ROT13 to a piece of text merely requires examining its alphabetic characters and replacing each one by the letter 13 places further along in the alphabet, wrapping back to the beginning if necessary

- Implement rot13 decoding
- decode this message:

  Zntargvp sebz bhgfvqr arne pbeare
  (from a geo-caching hint)

Files

Text Files

```
f = open('secrets.txt')
secret_data = f.read()
f.close()
```

`secret_data` is a string

(can also use `file()` – `open()` is preferred)

Files

Binary Files

```
f = open('secrets.txt', 'rb')
secret_data = f.read()
f.close()
```

`secret_data` is still a string

(with arbitrary bytes in it)

(See the `struct` module to unpack binary data)

Files

File Opening Modes

```
f = open('secrets.txt', [mode])
```

```
'r', 'w', 'a'
'rb', 'wb', 'ab'
r+, w+, a+
r+b, w+b, a+b
U
U+
```

Gotcha – w mode always clears the file

Text File Notes

Text is default

- Newlines are translated: `\r\n -> \n`
- – reading and writing!
- Use *nux-style in your code: `\n`
- Open text files with `'U'` "Universal" flag

Gotcha:

- no difference between text and binary on *nix
  - breaks on Windows

File Reading

Reading Part of a file

```
header_size = 4096

f = open('secrets.txt')
secret_data = f.read(header_size)
f.close()
```

File Reading

Common Idioms

```
for line in open('secrets.txt'):
    print line

f = open('secrets.txt')
while True:
    line = f.readline()
    if not line:
        break
    do_something_with_line()
```

File Writing

```
outfile = open('output.txt')

for i in range(10):
    outfile.write("this is line: %i\n"%i)
```

File Methods

Commonly Used Methods

```
f.read() f.readline()  f.readlines()

f.write(str) f.writelines(seq)

f.seek(offset)   f.tell()

f.flush()

f.close()
```

File Like Objects

File-like objects

Many classes implement the file interface:

- loggers
- `sys.stdout`
- `urllib.open()`
- pipes, subprocesses
- StringIO

`http://docs.python.org/library/stdtypes.html#bltin-file-objects`

StringIO

StringIO

```
In [417]: import StringIO
In [420]: f = StringIO.StringIO()

In [421]: f.write("somestuff")

In [422]: f.seek(0)

In [423]: f.read()
Out[423]: 'somestuff'
```

handy for testing

Exceptions

Another Branching structure:

```
try:
    do_something()
    f = open('missing.txt')
    process(f)   # never called if file missing
except IOError:
    print "couldn't open missing.txt"
```

Exceptions

Never Do this:

```
try:
    do_something()
    f = open('missing.txt')
    process(f)   # never called if file missing
except:
    print "couldn't open missing.txt"
```

Exceptions

Use Exceptions, rather than your own tests
– Don't do this:

```
do_something()
if os.path.exists('missing.txt'):
    f = open('missing.txt')
    process(f)   # never called if file missing
```

it will almost always work – but the almost will drive you crazy

Exceptions

"easier to ask forgiveness than permission"

– Grace Hopper

`http://www.youtube.com/watch?v=AZDWveIdqjY`

Exceptions

For simple scripts, let exceptions happen

Only handle the exception if the code can and will do something about it

(much better debugging info when an error does occur)

Exceptions – finally

```
try:
    do_something()
    f = open('missing.txt')
    process(f)   # never called if file missing
except IOError:
    print "couldn't open missing.txt"
finally:
    do_some_clean-up
```

the `finally:` clause will always run

Exceptions – else

```
try:
    do_something()
    f = open('missing.txt')
except IOError:
    print "couldn't open missing.txt"
else:
    process(f) # only called if there was no exception
```

Advantage:
you know where the Exception came from

Exceptions – using them

```
try:
    do_something()
    f = open('missing.txt')
except IOError as the_error:
    print the_error
    the_error.extra_info = "some more information"
    raise
```

Particularly useful if you catch more than one exception:

```
except (IOError, BufferError, OSError) as the_error:
    do_something_with (the_error)
```

Raising Exceptions

```
def divide(a,b):
    if b == 0:
        raise ZeroDivisionError("b can not be zero")
    else:
        return a / b
```

when you call it:

```
In [515]: divide (12,0)

ZeroDivisionError: b can not be zero
```

Built in Exceptions

You can create your own custom exceptions
But...

```
exp = \
 [name for name in dir(__builtin__) if "Error" in name]

len(exp)
32
```

For the most part, you can/should use a built in one

LAB
Exceptions Lab

Lightning Talk

Lighting Talk:

Joshua

Unicode

I hope you all read this:

The Absolute Minimum Every Software Developer Absolutely, Positively Must Know About Unicode and Character Sets (No Excuses!)

`http://www.joelonsoftware.com/articles/Unicode.html`

If not – go read it!

Unicode

Everything is Bytes

If it's on disk or on a network, it's bytes

Python provides some abstractions to make it easier to deal with bytes

Unicode

Unicode is a biggie

strings vs unicode
(`str()` vs. `unicode()`)

python 2.x vs 3.x

(actually, dealing with numbers rather than bytes is big – but we take that for granted)

Unicode

Strings are sequences of bytes

Unicode strings are sequences of platonic characters

Platonic characters cannot be written to disk or network!

(ANSI – one character == one byte – so easy!)

Unicode

the `unicode` object lets you work with characters

encoding is converting from a uncode object to bytes

decoding is converting from bytes to a unicode object

Unicode

```
import codecs
ord()
chr()
unichr()
str()
unicode()
encode()
decode()
```

Unicode Literals

1) Use unicode in your source files:

```
# -*- coding: utf-8 -*-
```

2) escape the unicode characters

```
print u"The integral sign: \u222B"
print u"The integral sign: \N{integral}"
```

lots of tables of code points online:
`http://inamidst.com/stuff/unidata/`

Unicode

Use unicode objects in all your code

decode on input

encode on output

Many packages do this for you
    (XML processing, databases, ...)

Gotcha:
    Python has a default encoding (usually ascii)

Unicode

Python Docs Unicode HowTo:

`http://docs.python.org/howto/unicode.html`

"Reading Unicode from a file is therefore simple:"

```
import codecs
f = codecs.open('unicode.rst', encoding='utf-8')
for line in f:
    print repr(line)
```

LAB

unicode exercises

- Find some nifty non-ascii characters you might use. Create a unicode object with them in two different ways.
- In the "code" dir for this week, there are two files:
  `text.utf16`
  `text.utf16`

Lightning Talk

Lighting Talk:

Bill

Paths

Relative paths:

`secret.txt`
`./secret.txt`

Absolute paths:

`/home/chris/secret.txt`

Either work with `open()`, etc.

(working directory only makes sense with command-line programs...)

os.path

```
os.getcwd() -- os.getcwdu()
chdir(path)

os.path.abspath()
os.path.relpath()
```

os.path

```
os.path.split()
os.path.splitext()
os.path.basename()
os.path.dirname()
os.path.join()
```

(all platform independent)

directories

`os.listdir()`
`os.mkdir()`

`os.walk()`

(higher level stuff in `shutil` module)

LAB

- write a program which prints the full path to all files in the current directory, one per line
- write a program which copies a file from a source, to a destination (without using shutil, or the OS copy command)
- update mail-merge from the previous lab to write output to individual files on disk
- advanced - implement your own iterator

Homework

- Finish (or re-factor) the Labs you didn't finish in class.