Mastering Python 3 I/O

David Beazley
http://www.dabeaz.com

Presented at PyCon'2010
Atlanta, Georgia

Copyright (C) 2010, David Beazley, http://www.dabeaz.com

This Tutorial

- It's about a very specific aspect of Python 3
- Maybe the most important part of Python 3
- Namely, the reimplemented I/O system

Copyright (C) 2010, David Beazley, http://www.dabeaz.com

Why I/O?

- Real programs interact with the world
  - They read and write files
  - They send and receive messages
  - They don't compute Fibonacci numbers
- I/O is at the heart of almost everything that Python is about (scripting, gluing, frameworks, C extensions, etc.)

Copyright (C) 2010, David Beazley, http://www.dabeaz.com

The I/O Problem

- Of all of the changes made in Python 3, it is my observation that I/O handling changes are the most problematic for porting
- Python 3 re-implements the entire I/O stack
- Python 3 introduces new programming idioms
- I/O handling issues can't be fixed by automatic code conversion tools (2to3)

Copyright (C) 2010, David Beazley, http://www.dabeaz.com

The Plan

- We're going to take a detailed top-to-bottom tour of the whole Python 3 I/O system
  - Text handling
  - Binary data handling
  - System interfaces
  - The new I/O stack
  - Standard library issues
  - Memory views, buffers, etc.

Copyright (C) 2010, David Beazley, http://www.dabeaz.com

Prerequisites

- I assume that you are already reasonably familiar with how I/O works in Python 2
  - str vs. unicode
  - print statement
  - open() and file methods
  - Standard library modules
  - General awareness of I/O issues
- Prior experience with Python 3 not required

Copyright (C) 2010, David Beazley, http://www.dabeaz.com

Performance Disclosure

- There are some performance tests
- Execution environment for tests:
  - 2.4 GHZ 4-Core MacPro, 3GB memory
  - OS-X 10.6.2 (Snow Leopard)
  - All Python interpreters compiled from source using same config/compiler
- Tutorial is not meant to be a detailed performance study so all results should be viewed as rough estimates

Copyright (C) 2010, David Beazley, http://www.dabeaz.com

Let's Get Started

- I have made a few support files:

http://www.dabeaz.com/python3io/index.html

- You can try some of the examples as we go
- However, it is fine to just watch/listen and try things on your own later

Copyright (C) 2010, David Beazley, http://www.dabeaz.com

Part 1

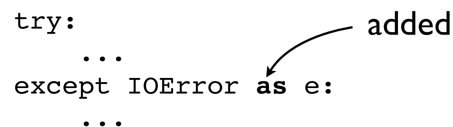
Introducing Python 3

Syntax Changes

- As you know, Python 3 changes syntax
- print is now a function print()

```
print("Hello World")
```

- Exception handling syntax changed slightly




```
try:
    ...
except IOError as e:
    ...
```

- Yes, your old code will break

Many New Features

- Python 3 introduces many new features
- Composite string formatting

```
"{0:10s} {1:10d} {2:10.2f}".format(name, shares, price)
```

- Dictionary comprehensions

```
a = {key.upper():value for key,value in d.items()}
```

- Function annotations

```
def square(x:int) -> int:
    return x*x
```

- Much more... but that's a different tutorial

Copyright (C) 2010, David Beazley, http://www.dabeaz.com

Changed Built-ins

- Many of the core built-in operations change
- Examples : range(), zip(), etc.

```
>>> a = [1,2,3]
>>> b = [4,5,6]
>>> c = zip(a,b)
>>> c
<zip object at 0x100452950>
>>>
```

- Typically related to iterators/generators

Copyright (C) 2010, David Beazley, http://www.dabeaz.com

Library Reorganization

- The standard library has been cleaned up
- Especially network/internet modules
- Example : Python 2

```
from urllib2 import urlopen
u = urlopen("http://www.python.org")
```

- Example : Python 3

```
from urllib.request import urlopen
u = urlopen("http://www.python.org")
```

Copyright (C) 2010, David Beazley, http://www.dabeaz.com

2to3 Tool

- There is a tool (2to3) that can be used to identify (and optionally fix) Python 2 code that must be changed to work with Python 3
- It's a command-line tool:

```
bash % 2to3 myprog.py
...
```

- Critical point : 2to3 can help, but it does not automate Python 2 to 3 porting

Copyright (C) 2010, David Beazley, http://www.dabeaz.com

2to3 Example

- Consider this Python 2 program

```
# printlinks.py
import urllib
import sys
from HTMLParser import HTMLParser

class LinkPrinter(HTMLParser):
    def handle_starttag(self,tag,attrs):
        if tag == 'a':
           for name,value in attrs:
               if name == 'href': print value

data = urllib.urlopen(sys.argv[1]).read()
LinkPrinter().feed(data)
```

- It prints all <a href="..."> links on a web page

Copyright (C) 2010, David Beazley, http://www.dabeaz.com

2to3 Example

- Here's what happens if you run 2to3 on it

```
bash % 2to3 printlinks.py
...
--- printlinks.py (original)
+++ printlinks.py (refactored)
@@ -1,12 +1,12 @@
-import urllib
+import urllib.request, urllib.parse, urllib.error
 import sys
-from HTMLParser import HTMLParser
+from html.parser import HTMLParser

 class LinkPrinter(HTMLParser):
     def handle_starttag(self,tag,attrs):
         if tag == 'a':
            for name,value in attrs:
-               if name == 'href': print value
+               if name == 'href': print(value)
...
```

It identifies lines that must be changed

Copyright (C) 2010, David Beazley, http://www.dabeaz.com

Fixed Code

- Here's an example of a fixed code (after 2to3)

```
import urllib.request, urllib.parse, urllib.error
import sys
from html.parser import HTMLParser

class LinkPrinter(HTMLParser):
    def handle_starttag(self,tag,attrs):
        if tag == 'a':
           for name,value in attrs:
               if name == 'href': print(value)

data = urllib.request.urlopen(sys.argv[1]).read()
LinkPrinter().feed(data)
```

- This is syntactically correct Python 3
- But, it still doesn't work. Do you see why?

Copyright (C) 2010, David Beazley, http://www.dabeaz.com

Broken Code

- Run it

```
bash % python3 printlinks.py http://www.python.org
Traceback (most recent call last):
  File "printlinks.py", line 12, in <module>
    LinkPrinter().feed(data)
  File "/Users/beazley/Software/lib/python3.1/html/parser.py",
line 107, in feed
    self.rawdata = self.rawdata + data
TypeError: Can't convert 'bytes' object to str implicitly
bash %
```

- That is an I/O handling problem
- Important lesson : <u>2to3 didn't find it</u>

Copyright (C) 2010, David Beazley, http://www.dabeaz.com

Actually Fixed Code

- This version works

```
import urllib.request, urllib.parse, urllib.error
import sys
from html.parser import HTMLParser

class LinkPrinter(HTMLParser):
    def handle_starttag(self,tag,attrs):
        if tag == 'a':
           for name,value in attrs:
               if name == 'href': print(value)

data = urllib.request.urlopen(sys.argv[1]).read()
LinkPrinter().feed(data.decode('utf-8'))
```

I added this one tiny bit (by hand)

Copyright (C) 2010, David Beazley, http://www.dabeaz.com

Important Lessons

- A lot of things change in Python 3
- 2to3 only fixes really "obvious" things
- It does not, in general, fix I/O problems
- Imagine applying it to a huge framework

Copyright (C) 2010, David Beazley, http://www.dabeaz.com

Part 2

Working with Text

Copyright (C) 2010, David Beazley, http://www.dabeaz.com

Making Peace with Unicode

- In Python 3, all text is Unicode
- All strings are Unicode
- All text-based I/O is Unicode
- You really can't ignore it or live in denial

Copyright (C) 2010, David Beazley, http://www.dabeaz.com

Unicode For Mortals

- I teach a lot of Python training classes
- I rarely encounter programmers who have a solid grasp on Unicode details (or who even care all that much about it to begin with)
- What follows : Essential details of Unicode that all Python 3 programmers must know
- You don't have to become a Unicode expert

Copyright (C) 2010, David Beazley, http://www.dabeaz.com

Text Representation

- Old-school programmers know about ASCII

	000	001	002	003	004	005	006	007
0	NUL 0000	DLE 0010	SP 0020	0 0030	@ 0040	P 0050	` 0060	p 0070
1	SOH 0001	DC1 0011	! 0021	1 0031	A 0041	Q 0051	a 0061	q 0071
2	STX 0002	DC2 0012	" 0022	2 0032	B 0042	R 0052	b 0062	r 0072
3	ETX 0003	DC3 0013	# 0023	3 0033	C 0043	S 0053	c 0063	s 0073
4	EOT 0004	DC4 0014	$ 0024	4 0034	D 0044	T 0054	d 0064	t 0074

- Each character has its own integer byte code
- Text strings are sequences of character codes

Copyright (C) 2010, David Beazley, http://www.dabeaz.com

Unicode Characters

- Unicode is the same idea only extended
- It defines a standard integer code for every character used in all languages (except for fictional ones such as Klingon, Elvish, etc.)
- The numeric value is known as a "code point"
- Typically denoted U+HHHH in conversation

```
ñ  = U+00F1
ε  = U+03B5
ઇ  = U+0A87
㌄  = U+3304
```

Copyright (C) 2010, David Beazley, http://www.dabeaz.com

Unicode Charts

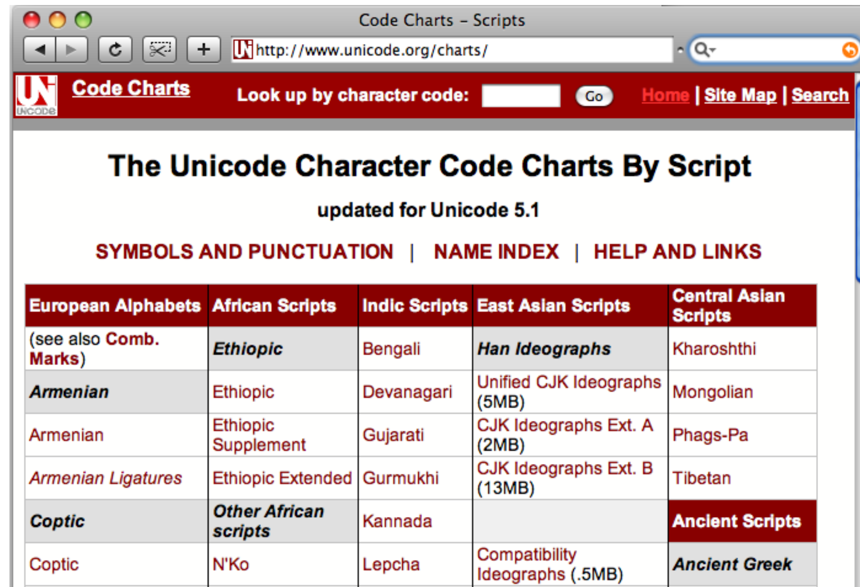
- A major problem : There are a lot of codes
- Largest supported code point U+10FFFF
- Code points are organized into charts

http://www.unicode.org/charts

- Go there and you will find charts organized by language or topic (e.g., greek, math, music, etc.)

Copyright (C) 2010, David Beazley, http://www.dabeaz.com

Unicode Charts



The Unicode Character Code Charts By Script

updated for Unicode 5.1

SYMBOLS AND PUNCTUATION | NAME INDEX | HELP AND LINKS

European Alphabets	African Scripts	Indic Scripts	East Asian Scripts	Central Asian Scripts
(see also **Comb. Marks**)	***Ethiopic***	Bengali	***Han Ideographs***	Kharoshthi
Armenian	Ethiopic	Devanagari	Unified CJK Ideographs (5MB)	Mongolian
Armenian	Ethiopic Supplement	Gujarati	CJK Ideographs Ext. A (2MB)	Phags-Pa
Armenian Ligatures	Ethiopic Extended	Gurmukhi	CJK Ideographs Ext. B (13MB)	Tibetan
Coptic	***Other African scripts***	Kannada		**Ancient Scripts**
Coptic	N'Ko	Lepcha	Compatibility Ideographs (.5MB)	***Ancient Greek***

Copyright (C) 2010, David Beazley, http://www.dabeaz.com

Unicode String Literals

- Strings can now contain any unicode character
- Example:

```
t = "That's a spicy jalapeño!"
                          ↑
```

- Problem : How do you indicate such characters?

Copyright (C) 2010, David Beazley, http://www.dabeaz.com

Using a Unicode Editor

- If you are using a Unicode-aware editor, you can type the characters in source code (save as UTF-8)

```
t = "That's a spicy Jalapeño!"
```

- Example : "Character & Keyboard" viewer (Mac)

Using Unicode Charts

- If you can't type it, use a code-point escape

0080 **C1 Controls and Latin-1 Supplement** **00FF**

	008	009	00A	00B	00C	00D	00E	00F
0	XXX 0080	DCS 0090	NB SP 00A0	° 00B0	À 00C0	Đ 00D0	à 00E0	ð 00F0
1	XXX 0081	PU1 0091	¡ 00A1	± 00B1	Á 00C1	Ñ 00D1	á 00E1	ñ 00F1

```
t = "That's a spicy Jalape\u00f1o!"
```

- \uxxxx - Embeds a Unicode code point in a string

Unicode Escapes

- There are three Unicode escapes
  - \xhh : Code points U+00 - U+FF
  - \uhhhh : Code points U+0100 - U+FFFF
  - \Uhhhhhhhh : Code points > U+10000
- Examples:

```
a = "\xf1"            # a = 'ñ'
b = "\u210f"          # b = 'ℏ'
c = "\U0001d122"      # c = '𝄢'
```

Copyright (C) 2010, David Beazley, http://www.dabeaz.com

Using Unicode Charts

- Code points also have descriptive names

00F1	ñ	LATIN SMALL LETTER N WITH TILDE
		≡ 006E n 0303 ◌̃
00F2	ò	LATIN SMALL LETTER O WITH GRAVE
		≡ 006F o 0300 ◌̀
00F3	ó	LATIN SMALL LETTER O WITH ACUTE
		≡ 006F o 0301 ◌́

- \N{name} - Embeds a named character

```
t = "Spicy Jalape\N{LATIN SMALL LETTER N WITH TILDE}o!"
```

Copyright (C) 2010, David Beazley, http://www.dabeaz.com

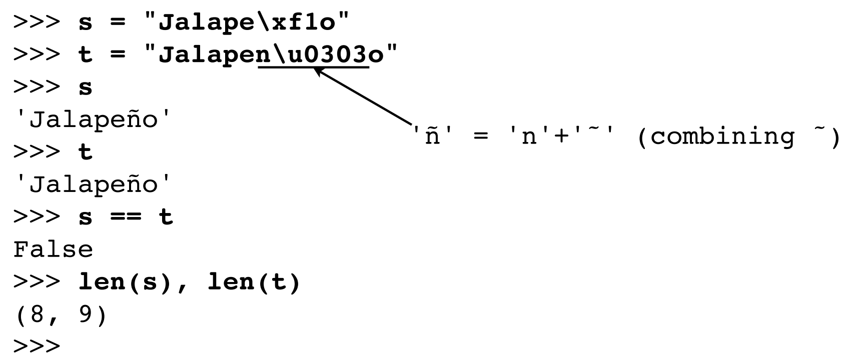
Commentary

- Don't overthink Unicode
- Unicode strings are mostly like ASCII strings except that there is a greater range of codes
- Everything that you normally do with strings (stripping, finding, splitting, etc.) still work, but are expanded

Copyright (C) 2010, David Beazley, http://www.dabeaz.com

A Caution

- Unicode is mostly like ASCII except when it's not



```
>>> s = "Jalape\xf1o"
>>> t = "Jalapen\u0303o"
>>> s
'Jalapeño'
>>> t
'Jalapeño'
>>> s == t
False
>>> len(s), len(t)
(8, 9)
>>>
```


- Many tricky bits if you get into internationalization
- However, that's a different tutorial

Copyright (C) 2010, David Beazley, http://www.dabeaz.com

Unicode Representation

- Internally, Unicode character codes are stored as multibyte integers (16 or 32 bits)

```
t = "Jalapeño"

004a 0061 006c 0061 0070 0065 00f1 006f   (UCS-2,16-bits)
0000004a 0000006a 0000006c 00000070 ...   (UCS-4,32-bits)
```

- You can find out using the sys module

```
>>> sys.maxunicode
65535                  # 16-bits

>>> sys.maxunicode
1114111                # 32-bits
```

- In C, it means a 'short' or 'int' is used

Copyright (C) 2010, David Beazley, http://www.dabeaz.com

Memory Use

- Yes, text strings in Python 3 require either 2x or 4x as much memory to store as Python 2
- For example: Read a 10MB ASCII text file

```
data = open("bigfile.txt").read()

>>> sys.getsizeof(data)          # Python 2.6
10485784

>>> sys.getsizeof(data)          # Python 3.1 (UCS-2)
20971578

>>> sys.getsizeof(data)          # Python 3.1 (UCS-4)
41943100
```

Copyright (C) 2010, David Beazley, http://www.dabeaz.com

Performance Impact

- Increased memory use does impact the performance of string operations that make copies of large substrings
  - Slices, joins, split, replace, strip, etc.
- Example:

```
timeit("text[:-1]","text='x'*100000")
```

Python 2.6.4 (bytes) : 11.5 s
Python 3.1.1 (UCS-2) : 24.1 s
Python 3.1.1 (UCS-4) : 47.1 s

- There are more bytes moving around

Copyright (C) 2010, David Beazley, http://www.dabeaz.com

Performance Impact

- Operations that process strings character often run at the same speed (or are faster)
  - lower, upper, find, regexs, etc.
- Example:

```
timeit("text.upper()","text='x'*1000")
```

Python 2.6.4 (bytes) : 9.3s
Python 3.1.1 (UCS-2) : 6.9s
Python 3.1.1 (UCS-4) : 7.0s

Copyright (C) 2010, David Beazley, http://www.dabeaz.com

Commentary

- Yes, text representation has an impact
- In your programs, you can work with text in the same way as you always have (text representation is just an internal detail)
- However, know that the performance may vary from 8-bit text strings in Python 2
- Study it if working with huge amounts of text

Copyright (C) 2010, David Beazley, http://www.dabeaz.com

Issue : Text Encoding

- The internal representation of characters is now almost never the same as how text is transmitted or stored in files

Text File

```
Hello World
```

File content (ASCII bytes)

```
48 65 6c 6c 6f 20 57 6f 72 6c 64 0a
```

read() write()

Python String Representation (UCS-4, 32-bit ints)

```
00000048 00000065 0000006c 0000006c
0000006f 00000020 00000057 0000006f
00000072 0000006c 00000064 0000000a
```

Copyright (C) 2010, David Beazley, http://www.dabeaz.com

Issue : Text Encoding

- There are also many possible file encodings for text (especially for non-ASCII)

	"Jalapeño"
latin-1	`4a 61 6c 61 70 65 f1 6f`
cp437	`4a 61 6c 61 70 65 a4 6f`
utf-8	`4a 61 6c 61 70 65 c3 b1 6f`
utf-16	`ff fe 4a 00 61 00 6c 00 61 00 70 00 65 00 f1 00 6f 00`

- Emphasize : They are only related to how text is stored in files, not stored in memory

Copyright (C) 2010, David Beazley, http://www.dabeaz.com

I/O Encoding

- All text is now encoded and decoded
- If reading text, it must be decoded from its source format into Python strings
- If writing text, it must be encoded into some kind of well-known output format
- This is a major difference between Python 2 and Python 3. In Python 2, you could write programs that just ignored encoding and read text as bytes (ASCII).

Copyright (C) 2010, David Beazley, http://www.dabeaz.com

Reading/Writing Text

- Built-in open() function has an optional encoding parameter

```
f = open("somefile.txt","rt",encoding="latin-1")
```

- If you omit the encoding, UTF-8 is assumed

```
>>> f = open("somefile.txt","rt")
>>> f.encoding
'UTF-8'
>>>
```

- Also, in case you're wondering, text file modes should be specified as "rt","wt","at", etc.

Copyright (C) 2010, David Beazley, http://www.dabeaz.com

Standard I/O

- Standard I/O streams also have encoding

```
>>> import sys
>>> sys.stdin.encoding
'UTF-8'
>>> sys.stdout.encoding
'UTF-8'
>>>
```

- Be aware that the encoding might change depending on the locale settings

```
>>> import sys
>>> sys.stdout.encoding
'US-ASCII'
>>>
```

Copyright (C) 2010, David Beazley, http://www.dabeaz.com

Binary File Modes

- Writing text on binary-mode files is an error

```
>>> f = open("foo.bin","wb")
>>> f.write("Hello World\n")
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: must be bytes or buffer, not str
>>>
```

- For binary I/O, Python 3 will never implicitly encode unicode strings and write them
- You must either use a text-mode file or explicitly encode (str.encode('*encoding*'))

Copyright (C) 2010, David Beazley, http://www.dabeaz.com

Important Encodings

- If you're not doing anything with Unicode (e.g., just processing ASCII files), there are still three encodings you should know
  - ASCII
  - Latin-1
  - UTF-8
- Will briefly describe each one

Copyright (C) 2010, David Beazley, http://www.dabeaz.com

ASCII Encoding

- Text that is restricted to 7-bit ASCII (0-127)
- Any characters outside of that range produce an encoding error

```
>>> f = open("output.txt","wt",encoding="ascii")
>>> f.write("Hello World\n")
12
>>> f.write("Spicy Jalapeño\n")
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
UnicodeEncodeError: 'ascii' codec can't encode
character '\xf1' in position 12: ordinal not in
range(128)
>>>
```

Copyright (C) 2010, David Beazley, http://www.dabeaz.com

Latin-1 Encoding

- Text that is restricted to 8-bit bytes (0-255)
- Byte values are left "as-is"

```
>>> f = open("output.txt","wt",encoding="latin-1")
>>> f.write("Spicy Jalapeño\n")
15
>>>
```

- Most closely emulates Python 2 behavior
- Also known as "iso-8859-1" encoding
- Pro tip: This is the fastest encoding for pure 8-bit text (ASCII files, etc.)

Copyright (C) 2010, David Beazley, http://www.dabeaz.com

UTF-8 Encoding

- A multibyte encoding that can represent all Unicode characters

```
Encoding                                  Description
0nnnnnnn                                  ASCII (0-127)
110nnnnn 10nnnnnn                         U+007F-U+07FF
1110nnnn 10nnnnnn 10nnnnnn                U+0800-U+FFFF
11110nnn 10nnnnnn 10nnnnnn 10nnnnnn       U+10000-U+10FFFF
```

- Example:

```
ñ = 0xf1 = 11110001

         = 11000011 10110001 = 0xc3 0xb1  (UTF-8)
```

Copyright (C) 2010, David Beazley, http://www.dabeaz.com

UTF-8 Encoding

- Main feature of UTF-8 is that ASCII is embedded within it
- If you're never working with international characters, UTF-8 will work transparently
- Usually a safe default to use when you're not sure (e.g., passing Unicode strings to operating system functions, interfacing with foreign software, etc.)

Copyright (C) 2010, David Beazley, http://www.dabeaz.com

Interlude

- If migrating from Python 2, keep in mind
  - Python 3 strings use multibyte integers
  - Python 3 always encodes/decodes I/O
  - If you don't say anything about encoding, Python 3 assumes UTF-8
- Everything that you did before should work just fine in Python 3 (probably)

Copyright (C) 2010, David Beazley, http://www.dabeaz.com

New Printing

- In Python 3, print() is used for text output
- Here is a mini porting guide

```
Python 2                Python 3

print x,y,z             print(x,y,z)
print x,y,z,            print(x,y,z,end=' ')
print >>f,x,y,z         print(x,y,z,file=f)
```

- However, print() has a few new tricks not available in Python 2

Copyright (C) 2010, David Beazley, http://www.dabeaz.com

Printing Enhancements

- Picking a different item separator

```
>>> print(1,2,3,sep=':')
1:2:3
>>> print("Hello","World",sep='')
HelloWorld
>>>
```

- Picking a different line ending

```
>>> print("What?",end="!?!\n")
What?!?!
>>>
```

- Relatively minor, but these features are often requested (e.g., "how do I get rid of the space?")

Copyright (C) 2010, David Beazley, http://www.dabeaz.com

Discussion : New Idioms

- In Python 2, you might have code like this

```
print ",".join([name,shares,price])
```

- Which of these is better in Python 3?

```
print(",".join([name,shares,price]))
```

- or -

```
print(name, shares, price, sep=",")
```

- Overall, I think I like the second one (even though it runs a little bit slower)

Copyright (C) 2010, David Beazley, http://www.dabeaz.com

New String Formatting

- Python 3 has completely revised formatting
- Here is old Python (%)

```
s = "%10s %10d %10.2f" % (name, shares, price)
```

- Here is Python 3

```
s = "{0:10s} {1:10d} {2:10.2f}".format(name,shares,price)
```

- You might find the new formatting jarring
- Let's talk about it

Copyright (C) 2010, David Beazley, http://www.dabeaz.com

First, Some History

- String formatting is one of the few features of Python 2 that can't be customized
- Classes can define __str__() and __repr__()
- However, they can't customize % processing
- Python 2.6/3.0 adds a __format__() special method that addresses this in conjunction with some new string formatting machinery

Copyright (C) 2010, David Beazley, http://www.dabeaz.com

String Conversions

- Objects now have three string conversions

```
>>> x = 1/3
>>> x.__str__()
'0.333333333333'
>>> x.__repr__()
'0.3333333333333333'
>>> x.__format__("0.2f")
'0.33'
>>> x.__format__("20.2f")
'                0.33'
>>>
```

- You will notice that __format__() takes a code similar to those used by the % operator

Copyright (C) 2010, David Beazley, http://www.dabeaz.com

format() function

- format(*obj*, *fmt*) calls __format__

```
>>> x = 1/3
>>> format(x,"0.2f")
'0.33'
>>> format(x,"20.2f")
'                0.33'
>>>
```

- This is analogous to str() and repr()

```
>>> str(x)
'0.333333333333'
>>> repr(x)
'0.3333333333333333'
>>>
```

Copyright (C) 2010, David Beazley, http://www.dabeaz.com

Format Codes (Builtins)

- For builtins, there are standard format codes

Old Format	New Format	Description
"%d"	"d"	Decimal Integer
"%f"	"f"	Floating point
"%s"	"s"	String
"%e"	"e"	Scientific notation
"%x"	"x"	Hexadecimal

- Plus there are some brand new codes

	"o"	Octal
	"b"	Binary
	"%"	Percent

Copyright (C) 2010, David Beazley, http://www.dabeaz.com

Format Examples

- Examples of simple formatting

```
>>> x = 42
>>> format(x,"x")
'2a'
>>> format(x,"b")
'101010'
>>> y = 2.71828
>>> format(y,"f")
'2.718280'
>>> format(y,"e")
'2.718280e+00'
>>> format(y,"%")
'271.828000%'
```

Copyright (C) 2010, David Beazley, http://www.dabeaz.com

Format Modifiers

- Field width and precision modifiers

```
[width][.precision]code
```

- Examples:

```
>>> y = 2.71828
>>> format(y,"0.2f")
'2.72'
>>> format(y,"10.4f")
'    2.7183'
>>>
```

- This is exactly the same convention as with the legacy % string formatting

Copyright (C) 2010, David Beazley, http://www.dabeaz.com

Alignment Modifiers

- Alignment Modifiers

```
[<|>|^][width][.precision]code

<   left align
>   right align
^   center align
```

- Examples:

```
>>> y = 2.71828
>>> format(y,"<20.2f")
'2.72                '
>>> format(y,"^20.2f")
'        2.72        '
>>> format(y,">20.2f")
'                2.72'
>>>
```

Copyright (C) 2010, David Beazley, http://www.dabeaz.com

Fill Character

- Fill Character

```
[fill][<|>|^][width][.precision]code
```

- Examples:

```
>>> x = 42
>>> format(x,"08d")
'00000042'
>>> format(x,"032b")
'00000000000000000000000000101010'
>>> format(x,"=^32d")
'===============42==============='
>>>
```

Copyright (C) 2010, David Beazley, http://www.dabeaz.com

Thousands Separator

- Insert a ',' before the precision specifier

```
[fill][<|>|^][width][,][.precision]code
```

- Examples:

```
>>> x = 123456789
>>> format(x,",d")
'123,456,789'
>>> format(x,"10,.2f")
'123,456,789.00'
>>>
```

- This is pretty new (see PEP 378)

Copyright (C) 2010, David Beazley, http://www.dabeaz.com

Discussion

- As you can see, there's a lot of flexibility in the new format method (there are other features not shown here)
- User-defined objects can also completely customize their formatting if they implement __format__(self,*fmt*)

Copyright (C) 2010, David Beazley, http://www.dabeaz.com

String .format() Method

- Strings have .format() method for formatting multiple values at once (replacement for %)

```
>>> "{0:10s} {1:10d} {2:10.2f}".format('ACME',50,91.10)
'ACME               50      91.10'
>>>
```

- format() method looks for formatting specifiers enclosed in { } and expands them
- Each {} is similar to a %fmt specifier with the old string formatting

Copyright (C) 2010, David Beazley, http://www.dabeaz.com

Format Specifiers

- Each specifier has the form : {*what:fmt*}
  - *what.* Indicates what is being formatted (refers to one of the arguments supplied to the format() method)
  - *fmt.* A format code. The same as what is supplied to the format() function
- Each {*what:fmt*} gets replaced by the result of format(*what,fmt*)

Copyright (C) 2010, David Beazley, http://www.dabeaz.com

Formatting Illustrated

- Arguments specified by position {*n:fmt*}

```
"{0:10s} {2:10.2f}".format('ACME',50,91.10)
```

- Arguments specified by keyword {*key:fmt*}

```
"{name:10s} {price:10.2f}".format(name='ACME',price=91.10)
```

- Arguments formatted in order {*:fmt*}

```
"{:10s} {:10d} {:10.2f}".format('ACME',50,91.10)
```

Copyright (C) 2010, David Beazley, http://www.dabeaz.com

Container Lookups

- You can index sequences and dictionaries

```
>>> stock = ('ACME',50,91.10)
>>> "{s[0]:10s} {s[2]:10.2f}".format(s=stock)
'ACME            91.10'

>>> stock = {'name':'ACME', 'shares':50, 'price':91.10 }
>>> "{0[name]:10s} {0[price]:10.2f}".format(stock)
'ACME            91.10'
>>>
```

- Restriction : You can't put arbitrary expressions in the [] lookup (has to be a number or simple string identifier)

Copyright (C) 2010, David Beazley, http://www.dabeaz.com

Attribute Access

- You can refer to instance attributes

```
class Stock(object):
    def __init__(self,name,shares,price):
        self.name = name
        self.shares = shares
        self.price  = price

>>> s = Stock('ACME',50,91.10)
>>> "{0.name:10s} {0.price:10.2f}".format(s)
'ACME            91.10'
>>>
```

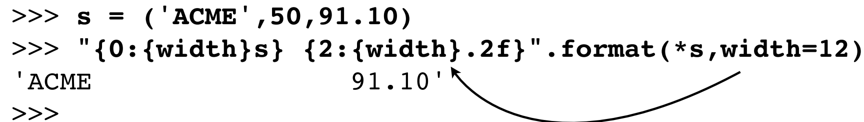
- Commentary : Nothing remotely like this with the old string formatting operator

Copyright (C) 2010, David Beazley, http://www.dabeaz.com

Nested Format Expansion

- .format() allows one level of nested lookups in the format part of each {}

```
>>> s = ('ACME',50,91.10)
>>> "{0:{width}s} {2:{width}.2f}".format(*s,width=12)
'ACME                 91.10'
>>>
```



- Probably best not to get too carried away in the interest of code readability though

Copyright (C) 2010, David Beazley, http://www.dabeaz.com

Other Formatting Details

- { and } must be escaped if part of formatting
- Use '{{ for '{'
- Use '}}' for '}'
- Example:

```
>>> "The value is {{{0}}}".format(42)
'The value is {42}'
>>>
```

Copyright (C) 2010, David Beazley, http://www.dabeaz.com

Commentary

- The new string formatting is very powerful
- However, I'll freely admit that it still feels very foreign to me (maybe it's due to my long history with using printf-style formatting)
- Python 3 still has the % operator, but it may go away some day (I honestly don't know).
- All things being equal, you probably want to embrace the new formatting

Copyright (C) 2010, David Beazley, http://www.dabeaz.com

Part 3

Binary Data Handling and Bytes

Copyright (C) 2010, David Beazley, http://www.dabeaz.com

Bytes and Byte Arrays

- Python 3 has support for "byte-strings"
- Two new types : bytes and bytearray
- They are quite different than Python 2 strings

Copyright (C) 2010, David Beazley, http://www.dabeaz.com

Defining Bytes

- Here's how to define byte "strings"

```
a = b"ACME 50 91.10"          # Byte string literal
b = bytes([1,2,3,4,5])        # From a list of integers
c = bytes(10)                 # An array of 10 zero-bytes
d = bytes("Jalapeño","utf-8") # Encoded from string
```

- Can also create from a string of hex digits

```
e = bytes.fromhex("48656c6c6f")
```

- All of these define an object of type "bytes"

```
>>> type(a)
<class 'bytes'>
>>>
```

- However, this new bytes object is an odd duck

Copyright (C) 2010, David Beazley, http://www.dabeaz.com

Bytes as Strings

- Bytes have standard "string" operations

```
>>> s = b"ACME 50 91.10"
>>> s.split()
[b'ACME', b'50', b'91.10']
>>> s.lower()
b'acme 50 91.10'
>>> s[5:7]
b'50'
```

- And bytes are immutable like strings

```
>>> s[0] = b'a'
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'bytes' object does not support item assignment
```

Copyright (C) 2010, David Beazley, http://www.dabeaz.com

Bytes as Integers

- Unlike Python 2, bytes are arrays of integers

```
>>> s = b"ACME 50 91.10"
>>> s[0]
65
>>> s[1]
67
>>>
```

- Same for iteration

```
>>> for c in s: print(c,end=' ')
65 67 77 69 32 53 48 32 57 49 46 49 48
>>>
```

- Hmmmm. Curious.

Copyright (C) 2010, David Beazley, http://www.dabeaz.com

bytearray objects

- A bytearray is a mutable bytes object

```
>>> s = bytearray(b"ACME 50 91.10")
>>> s[:4] = b"PYTHON"
>>> s
bytearray(b"PYTHON 50 91.10")
>>> s[0] = 0x70           # Must assign integers
>>> s
bytearray(b'pYTHON 50 91.10")
>>>
```

- It also gives you various list operations

```
>>> s.append(23)
>>> s.append(45)
>>> s.extend([1,2,3,4])
>>> s
bytearray(b'ACME 50 91.10\x17-\x01\x02\x03\x04')
>>>
```

Copyright (C) 2010, David Beazley, http://www.dabeaz.com

An Observation

- bytes and bytearray are not really meant to mimic Python 2 string objects
- They're closer to array.array('B',...) objects

```
>>> import array
>>> s = array.array('B',[10,20,30,40,50])
>>> s[1]
20
>>> s[1] = 200
>>> s.append(100)
>>> s.extend([65,66,67])
>>> s
array('B', [10, 200, 30, 40, 50, 100, 65, 66, 67])
>>>
```

Copyright (C) 2010, David Beazley, http://www.dabeaz.com

Bytes and Strings

- Bytes are not meant for text processing
- In fact, if you try to use them for text, you will run into weird problems
- Python 3 strictly separates text (unicode) and bytes everywhere
- This is probably the most major difference between Python 2 and 3.

Copyright (C) 2010, David Beazley, http://www.dabeaz.com

Mixing Bytes and Strings

- Mixed operations fail miserably

```
>>> s = b"ACME 50 91.10"
>>> 'ACME' in s
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: Type str doesn't support the buffer API
>>>
```

- Huh?!?? Buffer API?
- We'll cover that later...

Copyright (C) 2010, David Beazley, http://www.dabeaz.com

Printing Bytes

- Printing and text-based I/O operations do not work in a useful way with bytes

```
>>> s = b"ACME 50 91.10"
>>> print(s)
b'ACME 50 91.10'
>>>
```

Notice the leading b' and trailing quote in the output.

- There's no way to fix this. print() should only be used for outputting text (unicode)

Copyright (C) 2010, David Beazley, http://www.dabeaz.com

Formatting Bytes

- Bytes do not support operations related to formatted output (%, .format)

```
>>> s = b"%0.2f" % 3.14159
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unsupported operand type(s) for %: 'bytes' and
'float'
>>>
```

- So, just forget about using bytes for any kind of useful text output, printing, etc.
- No, seriously.

Copyright (C) 2010, David Beazley, http://www.dabeaz.com

Commentary

- Why am I focusing on this "bytes as text" issue?
- If you are writing scripts that do simple ASCII text processing, you might be inclined to use bytes as a way to avoid the overhead of Unicode
- You might think that bytes are exactly the same as the familiar Python 2 string object
- This is wrong. Bytes are not text. Using bytes as text will lead to convoluted non-idiomatic code

Copyright (C) 2010, David Beazley, http://www.dabeaz.com

How to Use Bytes

- To use the bytes objects, focus on problems related to low-level I/O handling (message passing, distributed computing, etc.)
- I will show some examples that illustrate
- A complaint: documentation (online and books) is extremely thin on explaining practical uses of bytes and bytearray objects
- Hope to rectify that a little bit here

Copyright (C) 2010, David Beazley, http://www.dabeaz.com

Example : Reassembly

- In Python 2, you may know that string concatenation leads to bad performance

```
msg = ""
while True:
   chunk = s.recv(BUFSIZE)
   if not chunk:
       break
   msg += chunk
```

- Here's the common workaround (hacky)

```
chunks = []
while True:
   chunk = s.recv(BUFSIZE)
   if not chunk:
       break
   chunks.append(chunk)
msg = b"".join(chunks)
```

Copyright (C) 2010, David Beazley, http://www.dabeaz.com

Example : Reassembly

- Here's a new approach in Python 3

```
msg = bytearray()
while True:
   chunk = s.recv(BUFSIZE)
   if not chunk:
       break
   msg.extend(chunk)
```

- You treat the bytearray as a list and just append/extend new data at the end as you go
- I like it. It's clean and intuitive.

Copyright (C) 2010, David Beazley, http://www.dabeaz.com

Example: Reassembly

- The performance is good too
- Concat 1024 32-byte chunks together (10000x)

Concatenation	:	18.49s
Joining	:	1.55s
Extending a bytearray	:	1.78s

- There are many parts of the Python standard library that might benefit (e.g., ByteIO objects, WSGI, multiprocessing, pickle, etc.)

Copyright (C) 2010, David Beazley, http://www.dabeaz.com

Example: Record Packing

- Suppose you wanted to use the struct module to incrementally pack a large binary message

```
objs = [ ... ]          # List of tuples to pack
msg = bytearray()       # Empty message

# First pack the number of objects
msg.extend(struct.pack("<I",len(objs)))

# Incrementally pack each object
for x in objs:
    msg.extend(struct.pack(fmt, *x))

# Do something with the message
f.write(msg)
```

- I like this as well.

Copyright (C) 2010, David Beazley, http://www.dabeaz.com

Comment : Writes

- The previous example is one way to avoid making lots of small write operations
- Instead you collect data into one large message that you output all at once.
- Improves I/O performance and code is nice

Copyright (C) 2010, David Beazley, http://www.dabeaz.com

Example : Calculations

- Run a byte array through an XOR-cipher

```
>>> s = b"Hello World"
>>> t = bytes(x^42 for x in s)
>>> t
b'bOFFE\n}EXFN'
>>> bytes(x^42 for x in t)
b'Hello World'
>>>
```

- Compute and append a LRC checksum to a msg

```
# Compute the checksum and append at the end
chk = 0
for n in msg:
    chk ^= n
msg.append(chk)
```

Copyright (C) 2010, David Beazley, http://www.dabeaz.com

Commentary

- I'm excited about the new bytearray object
- Many potential uses in building low-level infrastructure for networking, distributed computing, messaging, embedded systems, etc.
- May make much of that code cleaner, faster, and more memory efficient
- Still more features to come...

Copyright (C) 2010, David Beazley, http://www.dabeaz.com

Part 4

System Interfaces

Copyright (C) 2010, David Beazley, http://www.dabeaz.com

System Interfaces

- Major parts of the Python library are related to low-level systems programming, sysadmin, etc.
  - os, os.path, glob, subprocess, socket, etc.
- Unfortunately, there are some really sneaky aspects of using these modules with Python 3
- It concerns the Unicode/Bytes separation

Copyright (C) 2010, David Beazley, http://www.dabeaz.com

The Problem

- To carry out system operations, the Python interpreter executes standard C system calls
- For example, POSIX calls on Unix

```
int fd = open(filename, O_RDONLY);
```

- However, names used in system interfaces (e.g., filenames, program names, etc.) are specified as byte strings (char *)
- Bytes also used for environment variables and command line options

Copyright (C) 2010, David Beazley, http://www.dabeaz.com

Question

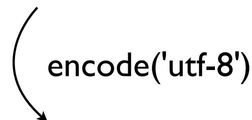
- How does Python 3 integrate strings (Unicode) with byte-oriented system interfaces?
- Examples:
  - Filenames
  - Command line arguments (sys.argv)
  - Environment variables (os.environ)
- Note: You should care about this if you use Python for various system tasks

Copyright (C) 2010, David Beazley, http://www.dabeaz.com

Name Encoding

- Standard practice is for Python 3 to UTF-8 encode all names passed to system calls

Python :
```
f = open("somefile.txt","wt")
```



C/syscall :
```
open("somefile.txt",O_WRONLY)
```

- This is usually a safe bet
- ASCII is a subset and UTF-8 is an extension that most operating systems support

Copyright (C) 2010, David Beazley, http://www.dabeaz.com

Arguments & Environ

- Similarly, Python decodes arguments and environment variables using UTF-8

Python 3:

```
bash % python foo.py arg1 arg2 ...   ------------>  sys.argv
                                     decode('utf-8')

TERM=xterm-color
SHELL=/bin/bash
USER=beazley
PATH=/usr/bin:/bin:/usr/sbin:...     ------------>  os.environ
LANG=en_US.UTF-8                     decode('utf-8')
HOME=/Users/beazley
LOGNAME=beazley
...
```

Copyright (C) 2010, David Beazley, http://www.dabeaz.com

Lurking Danger

- Be aware that some systems accept, but do not strictly enforce UTF-8 encoding of names
- This is extremely subtle, but it means that names used in system interfaces don't necessarily match the encoding that Python 3 wants
- Will show a pathological example to illustrate

Copyright (C) 2010, David Beazley, http://www.dabeaz.com

Example : A Bad Filename

- Start Python 2.6 on Linux and create a file using the open() function like this:

```
>>> f = open("jalape\xf1o.txt","w")
>>> f.write("Bwahahahaha!\n")
>>> f.close()
```

- This creates a file with a single non-ASCII byte (\xf1, 'ñ') embedded in the filename
- The filename is not UTF-8, but it still "works"
- Question: What happens if you try to do something with that file in Python 3?

Copyright (C) 2010, David Beazley, http://www.dabeaz.com

Example : A Bad Filename

- Python 3 won't be able to open the file

```
>>> f = open("jalape\xf1o.txt")
Traceback (most recent call last):
...
IOError: [Errno 2] No such file or directory: 'jalapeño.txt'
>>>
```

- This is caused by an encoding mismatch

```
"jalape\xf1o.txt"
        | UTF-8
        v
b"jalape\xc3\xb1o.txt"
        | open()
        v
      Fails!
```

It fails because this is the actual filename

```
b"jalape\xf1o.txt"
```

Copyright (C) 2010, David Beazley, http://www.dabeaz.com

Example : A Bad Filename

- Bad filenames cause weird behavior elsewhere
  - Directory listings
  - Filename globbing
- Example : What happens if a non UTF-8 name shows up in a directory listing?
- In early versions of Python 3, such names were silently discarded (made invisible). Yikes!

Copyright (C) 2010, David Beazley, http://www.dabeaz.com

Names as Bytes

- You can specify filenames using byte strings instead of strings as a workaround

```
>>> f = open(b"jalape\xf1o.txt")
>>>
```

Notice bytes

```
>>> files = glob.glob(b"*.txt")
>>> files
[b'jalape\xf1o.txt', b'spam.txt']
>>>
```

- This turns off the UTF-8 encoding and returns all results as bytes
- Note: Not obvious and a little hacky

Copyright (C) 2010, David Beazley, http://www.dabeaz.com

Surrogate Encoding

- In Python 3.1, non-decodable (bad) characters in filenames and other system interfaces are translated using "surrogate encoding" as described in PEP 383.
- This is a Python-specific "trick" for getting characters that don't decode as UTF-8 to pass through system calls in a way where they still work correctly

Copyright (C) 2010, David Beazley, http://www.dabeaz.com

Surrogate Encoding

- Idea : Any non-decodable bytes in the range 0x80-0xff are translated to Unicode characters U+DC80-U+DCFF
- Example:

```
b"jalape\xf1o.txt"
        |  surrogate encoding
        v
"jalape\udcf1o.txt"
```

- Similarly, Unicode characters U+DC80-U+DCFF are translated back into bytes 0x80-0xff when presented to system interfaces

Copyright (C) 2010, David Beazley, http://www.dabeaz.com

Surrogate Encoding

- You will see this used in various library functions and it works for functions like open()
- Example:

```
>>> glob.glob("*.txt")
[ 'jalape\udcf1o.txt', 'spam.txt']
```

notice the odd unicode character

```
>>> f = open("jalape\udcf1o.txt")
>>>
```

- If you ever see a \udcxx character, it means that a non-decodable byte was passed in from a system interface

Copyright (C) 2010, David Beazley, http://www.dabeaz.com

Surrogate Encoding

- Question : Does this break part of Unicode?
- Answer : Unsure
- This uses a range of Unicode dedicated for a feature known as "surrogate pairs". A pair of Unicode characters encoded like this

```
(U+D800-U+DBFF, U+DC00-U+DFFF)
```

- In Unicode, you would never see a U+DCxx character appearing all on its own

Copyright (C) 2010, David Beazley, http://www.dabeaz.com

Caution : Printing

- Non-decodable bytes will break print()

```
>>> files = glob.glob("*.txt")
>>> files
[ 'jalape\udcf1o.txt', 'spam.txt']
>>> for name in files:
...     print(name)
...
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
UnicodeEncodeError: 'utf-8' codec can't encode character
'\udcf1' in position 6: surrogates not allowed
>>>
```

- Arg! If you're using Python for file manipulation or system administration you need to be careful

Copyright (C) 2010, David Beazley, http://www.dabeaz.com

Implementation

- Surrogate encoding is implemented as an error handler for encode() and decode()
- Example:

```
>>> s = b"jalape\xf1o.txt"
>>> t = s.decode('utf-8','surrogateescape')
>>> t
'jalape\udcf1o.txt'

>>> t.encode('utf-8','surrogateescape')
b'jalape\xf1o.txt'
>>>
```

- If you are porting code that deals with system interfaces, you might need to do this

Copyright (C) 2010, David Beazley, http://www.dabeaz.com

Commentary

- This handling of Unicode in system interfaces is also of interest to C/C++ extensions
- What happens if a C/C++ function returns an improperly encoded byte string?
- What happens in ctypes? Swig?
- Seems unexplored (too obscure? new?)

Copyright (C) 2010, David Beazley, http://www.dabeaz.com

Part 5

The io module

Copyright (C) 2010, David Beazley, http://www.dabeaz.com

I/O Implementation

- I/O in Python 2 is largely based on C I/O
- For example, the "file" object is just a thin layer over a C "FILE *" object
- Python 3 changes this
- In fact, Python 3 has a complete ground-up reimplementation of the whole I/O system

Copyright (C) 2010, David Beazley, http://www.dabeaz.com

The open() function

- For files, you still use open() as you did before
- However, the result of calling open() varies depending on the file mode and buffering
- Carefully study the output of this:

```
>>> open("foo.txt","rt")
<_io.TextIOWrapper name='foo.txt' encoding='UTF-8'>
>>> open("foo.txt","rb")
<_io.BufferedReader name='foo.txt'>
>>> open("foo.txt","rb",buffering=0)
<_io.FileIO name='foo.txt' mode='rb'>
>>>
```

Notice how you're getting a different kind of result here

Copyright (C) 2010, David Beazley, http://www.dabeaz.com

The io module

- The core of the I/O system is implemented in the io library module
- It consists of a collection of different I/O classes

```
FileIO
BufferedReader
BufferedWriter
BufferedRWPair
BufferedRandom
TextIOWrapper
BytesIO
StringIO
```

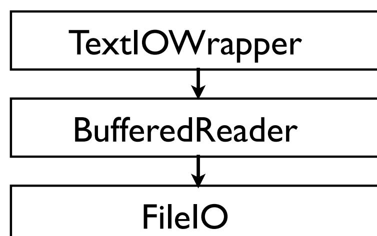
- Each class implements a different kind of I/O
- The classes get layered to add features

Copyright (C) 2010, David Beazley, http://www.dabeaz.com

Layering Illustrated

- Here's the result of opening a "text" file

```
open("foo.txt","rt")
```



- Keep in mind: This is very different from Python 2
- Inspired by Java? (don't know, maybe)

Copyright (C) 2010, David Beazley, http://www.dabeaz.com

FileIO Objects

- An object representing raw unbuffered binary I/O
- FileIO(*name* [, *mode* [, *closefd*])

  name : Filename or integer fd
  mode : File mode ('r', 'w', 'a', 'r+',etc.)
  closefd : Flag that controls whether close() called

- Under the covers, a FileIO object is directly layered on top of operating system functions such as read(), write()

Copyright (C) 2010, David Beazley, http://www.dabeaz.com

FileIO Usage

- FileIO replaces os module functions
- Example : Python 2 (os module)

```
fd = os.open("somefile",os.O_RDONLY)
data = os.read(fd,4096)
os.lseek(fd,16384,os.SEEK_SET)
...
```

- Example : Python 3 (FileIO object)

```
f = io.FileIO("somefile","r")
data = f.read(4096)
f.seek(16384,os.SEEK_SET)
...
```

- It's a low-level file with a file-like interface (nice)

Copyright (C) 2010, David Beazley, http://www.dabeaz.com

Direct System I/O

- FileIO directly exposes the behavior of low-level system calls on file descriptors
- This includes:
  - Partial read/writes
  - Returning system error codes
  - Blocking/nonblocking I/O handling
- System hackers want this

Copyright (C) 2010, David Beazley, http://www.dabeaz.com

Direct System I/O

- File operations (read/write) execute a single system call no matter what

```
data = f.read(8192)      # Executes one read syscall
f.write(data)            # Executes one write syscall
```

- This might mean partial data (you must check)

Copyright (C) 2010, David Beazley, http://www.dabeaz.com

Commentary

- FileIO is the most critical object in the I/O stack
- Everything else depends on it
- Nothing quite like it in Python 2

Copyright (C) 2010, David Beazley, http://www.dabeaz.com

BufferedIO Objects

- The following classes implement buffered I/O

```
BufferedReader(f [, buffer_size])
BufferedWriter(f [, buffer_size [, max_buffer_size]])
BufferedRWPair(f_read, f_write
                 [, buffer_size [, max_buffer_size]])
BufferedRandom(f [, buffer_size [, max_buffer_size]])
```

- Each of these classes is layered over a supplied raw FileIO object (*f*)

```
f = io.FileIO("foo.txt")      # Open the file (raw I/O)
g = io.BufferedReader(f)      # Put buffering around it

f = io.BufferedReader(io.FileIO("foo.txt")) # Alternative
```

Copyright (C) 2010, David Beazley, http://www.dabeaz.com

Buffered Operations

- Buffered readers implement these methods

```
f.peek([n])      # Return up to n bytes of data without
                 # advancing the file pointer

f.read([n])      # Return n bytes of data as bytes

f.read1([n])     # Read up to n bytes using a single
                 # read() system call
```

- Buffered writers implement these methods

```
f.write(bytes)  # Write bytes
f.flush()       # Flush output buffers
```

- Other ops (seek, tell, close, etc.) work as well

Copyright (C) 2010, David Beazley, http://www.dabeaz.com

TextIOWrapper

- The object that implements text-based I/O

```
TextIOWrapper(buffered [, encoding [, errors
                       [, newline [, line_buffering]]]])

buffered         - A buffered file object
encoding         - Text encoding (e.g., 'utf-8')
errors           - Error handling policy (e.g. 'strict')
newline          - '', '\n', '\r', '\r\n', or None
line_buffering   - Flush output after each line (False)
```

- It is layered on a buffered I/O stream

```
f = io.FileIO("foo.txt")        # Open the file (raw I/O)
g = io.BufferedReader(f)        # Put buffering around it
h = io.TextIOWrapper(g,"utf-8") # Text I/O wrapper
```

Copyright (C) 2010, David Beazley, http://www.dabeaz.com

TextIOWrapper and codecs

- Python 2 used the codecs module for unicode
- TextIOWrapper It is a completely new object, written almost entirely in C
- It kills codecs.open() in performance

```
for line in open("biglog.txt",encoding="utf-8"):
    pass
```

3.8 sec

```
f = codecs.open("biglog.txt",encoding="utf-8")
for line in f:
    pass
```

53.3 sec

Note: both tests performed using Python-3.1.1

Copyright (C) 2010, David Beazley, http://www.dabeaz.com

Putting it All Together

- As a user, you don't have to worry too much about how the different parts of the I/O system are put together (all of the different classes)
- The built-in open() function constructs the proper set of IO objects depending on the supplied parameters
- Power users might use the io module directly for more precise control over special cases

Copyright (C) 2010, David Beazley, http://www.dabeaz.com

open() Revisited

- Here is the full prototype

```
open(name [, mode [, buffering [, encoding [, errors
          [, newline [, closefd]]]]]])
```

- The different parameters get passed to underlying objects that get created

name, *mode*, *closefd* → FileIO

buffering → BufferedReader, BufferedWriter

encoding, *errors*, *newline* → TextIOWrapper

Copyright (C) 2010, David Beazley, http://www.dabeaz.com

open() Revisited

- The type of IO object returned depends on the supplied mode and buffering parameters

mode	buffering	Result
any binary	0	FileIO
"rb"	!= 0	BufferedReader
"wb","ab"	!= 0	BufferedWriter
"rb+","wb+","ab+"	!= 0	BufferedRandom
any text	!= 0	TextIOWrapper

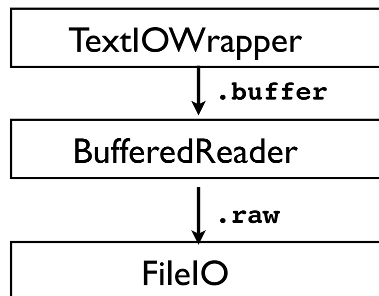
- Note: Certain combinations are illegal and will produce an exception (e.g., unbuffered text)

Copyright (C) 2010, David Beazley, http://www.dabeaz.com

Unwinding the I/O Stack

- Sometimes you might need to unwind a file

```
open("foo.txt","rt")
```



- Scenario : You were given an open text-mode file, but want to use it in binary mode

Copyright (C) 2010, David Beazley, http://www.dabeaz.com

I/O Performance

- Question : How does new I/O perform?
- Will compare:
  - Python 2.6.4 built-in open()
  - Python 3.1.1 built-in open()
- Note: This is not exactly a fair test--the Python 3 open() has to decode Unicode text
- However, it's realistic, because most programmers use open() without thinking about it

Copyright (C) 2010, David Beazley, http://www.dabeaz.com

I/O Performance

- Read a 100 Mbyte text file all at once

```
data = open("big.txt").read()
```

Python 2.6.4	: 0.16s	
Python 3.1 (UCS-2, UTF-8)	: 0.95s	
Python 3.1 (UCS-4, UTF-8)	: 1.67s	

Yes, you get overhead due to text decoding

- Read a 100 Mbyte binary file all at once

```
data = open("big.bin","rb").read()
```

Python 2.6.4	: 0.16s
Python 3.1 (UCS-2, UTF-8)	: 0.16s
Python 3.1 (UCS-4, UTF-8)	: 0.16s

(I couldn't observe any noticeable difference)

- Note: tests conducted with warm disk cache

Copyright (C) 2010, David Beazley, http://www.dabeaz.com

I/O Performance

- Write a 100 Mbyte text file all at once

```
open("foo.txt","wt").write(text)
```

Python 2.6.4	: 2.30s
Python 3.1 (UCS-2, UTF-8)	: 2.47s
Python 3.1 (UCS-4, UTF-8)	: 2.55s

- Write a 100 Mbyte binary file all at once

```
data = open("big.bin","wb").write(data)
```

Python 2.6.4	: 2.16s
Python 3.1 (UCS-2, UTF-8)	: 2.16s
Python 3.1 (UCS-4, UTF-8)	: 2.16s

(I couldn't observe any noticeable difference)

- Note: tests conducted with warm disk cache

Copyright (C) 2010, David Beazley, http://www.dabeaz.com

I/O Performance

- Iterate over 730000 lines of a big log file (text)

```
for line in open("biglog.txt"):
    pass
```

Python 2.6.4 : 0.24s
Python 3.1 (UCS-2, UTF-8) : 0.57s
Python 3.1 (UCS-4, UTF-8) : 0.82s

- Iterate over 730000 lines of a log file (binary)

```
for line in open("biglog.txt","rb"):
    pass
```

Python 2.6.4 : 0.24s
Python 3.1 (UCS-2, UTF-8) : 0.29s
Python 3.1 (UCS-4, UTF-8) : 0.29s

Copyright (C) 2010, David Beazley, http://www.dabeaz.com

I/O Performance

- Write 730000 lines log data (text)

```
open("biglog.txt","wt").writelines(lines)
```

Python 2.6.4 : 1.3s
Python 3.1 (UCS-2, UTF-8) : 1.4s
Python 3.1 (UCS-4, UTF-8) : 1.4s

Note: higher variance in observed times. These are 10 sample averages (rough ballpark)

- Write 730000 lines of log data (binary)

```
for line in open("biglog.txt","wb"):
    pass
```

Python 2.6.4 : 1.3s
Python 3.1 (UCS-2, UTF-8) : 1.3s
Python 3.1 (UCS-4, UTF-8) : 1.3s

Copyright (C) 2010, David Beazley, http://www.dabeaz.com

Commentary

- For binary, the Python 3 I/O system is comparable to Python 2 in performance
- Text based I/O has an unavoidable penalty
  - Extra decoding (UTF-8)
  - An extra memory copy
- You might be able to minimize the decoding penalty by specifying 'latin-1' (fastest)
- The memory copy can't be eliminated

Copyright (C) 2010, David Beazley, http://www.dabeaz.com

Commentary

- Reading/writing always involves bytes

```
"Hello World" -> 48 65 6c 6c 6f 20 57 6f 72 6c 64
```

- To get it to Unicode, it has to be copied to multibyte integers (no workaround)

```
48 65 6c 6c 6f 20 57 6f 72 6c 64
```

Unicode conversion

```
0048 0065 006c 006c 006f 0020 0057 006f 0072 006c 0064
```

- The only way to avoid this is to never convert bytes into a text string (not always practical)

Copyright (C) 2010, David Beazley, http://www.dabeaz.com

Advice

- Heed the advice of the optimization gods---ask yourself if it's really worth worrying about (premature optimization as the root of all evil)
- No seriously... does it matter for your app?
- If you are processing huge (no, gigantic) amounts of 8-bit text (ASCII, Latin-1, UTF-8, etc.) and I/O has been determined to be the bottleneck, there is one approach to optimization that might work

Copyright (C) 2010, David Beazley, http://www.dabeaz.com

Text Optimization

- Perform all I/O in binary/bytes and defer Unicode conversion to the last moment
- If you're filtering or discarding huge parts of the text, you might get a big win
- Example : Log file parsing

Copyright (C) 2010, David Beazley, http://www.dabeaz.com

Example

- Find all URLs that 404 in an Apache log

```
140.180.132.213 - - [...] "GET /ply/ply.html HTTP/1.1" 200 97238
140.180.132.213 - - [...] "GET /favicon.ico HTTP/1.1" 404 133
```

- Processing everything as text

```
error_404_urls = set()
for line in open("biglog.txt"):
    fields = line.split()
    if fields[-2] == '404':
        error_404_urls.add(fields[-4])

for name in error_404_urls:
    print(name)
```

Python 2.6.4 : 1.21s
Python 3.1 (UCS-2) : 2.12s
Python 3.1 (UCS-4) : 2.56s

Copyright (C) 2010, David Beazley, http://www.dabeaz.com

Example Optimization

- Deferred text conversion

```
error_404_urls = set()
for line in open("biglog.txt","rb"):
    fields = line.split()
    if fields[-2] == b'404':
        error_404_urls.add(fields[-4])

for name in error_404_urls:
    print(name.decode('latin-1'))
```

Unicode conversion here

Python 2.6.4 : 1.21s
Python 3.1 (UCS-2) : 1.21s
Python 3.1 (UCS-4) : 1.26s

Copyright (C) 2010, David Beazley, http://www.dabeaz.com

Part 6

Standard Library Issues

Copyright (C) 2010, David Beazley, http://www.dabeaz.com

Text, Bytes, and the Library

- In Python 2, you could be sloppy about the distinction between text and bytes in many library functions
    - Networking modules
    - Data handling modules
    - Various sorts of conversions
- In Python 3, you must be very precise

Copyright (C) 2010, David Beazley, http://www.dabeaz.com

Example : Socket Sends

- Here's a skeleton of some sloppy Python 2 code

```
def send_response(s,code,msg):
    s.sendall("HTTP/1.0 %s %s\r\n" % (code,msg))

send_response(s,"200","OK")
```

- This is almost guaranteed to break
- Reason : Almost every library function that communicates with the outside world (sockets, urllib, SocketServer, etc.) now uses binary I/O
- So, text operations are going to fail

Copyright (C) 2010, David Beazley, http://www.dabeaz.com

Example : Socket Sends

- In Python 3, you must explicitly encode text

```
def send_response(s,code,msg):
    resp = "HTTP/1.0 {:s} {:s}\r\n".format(code,msg)
    s.sendall(resp.encode('ascii'))

send_response(s,"200","OK")
```

- Commentary : You really should have been doing this in Python 2 all along

Copyright (C) 2010, David Beazley, http://www.dabeaz.com

Rules of Thumb

- All incoming text data must be decoded

```
rawmsg = s.recv(16384)        # Read from a socket
msg = rawmsg.decode('utf-8')  # Decode
...
```

- All outgoing text data must be encoded

```
rawmsg = msg.encode('ascii')
s.send(rawmsg)
...
```

- Code most affected : anything that's directly working with low-level network protocols (HTTP, SMTP, FTP, etc.)

Copyright (C) 2010, David Beazley, http://www.dabeaz.com

Tricky Text Conversions

- Certain "text" conversions in the library do not produce unicode text strings
  - Base 64, quopri, binascii
- Example:

```
>>> a = b"Hello"
>>> print(binascii.b2a_hex(a))
b'48656c6c6f'
>>> print(base64.b64encode(a))
b'SGVsbG8='
>>>
```

bytes (arrows pointing to b'48656c6c6f' and b'SGVsbG8=')

- Need to be careful if using these to embed data in text file formats (e.g., XML, JSON, etc.)

Copyright (C) 2010, David Beazley, http://www.dabeaz.com

Commentary

- When updating the Python Essential Reference to cover Python 3 features, byte/string issues in the standard library were one of the most frequently encountered problems
- Documentation not updated to correctly to indicate the requirement of bytes
- Various bugs in network/internet related code due to byte/string separation

Copyright (C) 2010, David Beazley, http://www.dabeaz.com

Part 7

Memory Views and I/O

Copyright (C) 2010, David Beazley, http://www.dabeaz.com

Memory Buffers

- Many objects in Python consist of contiguously allocated memory regions
  - Byte strings and byte arrays
  - Arrays (created by array module)
  - ctypes arrays/structures
  - Numpy arrays (not py3k yet)
- These objects have a special relationship with the I/O system

Copyright (C) 2010, David Beazley, http://www.dabeaz.com

Direct I/O with Buffers

- Objects consisting of contiguous memory regions can be used with I/O operations without making extra buffer copies

- reads and writes can be made to work directly with the underlying memory buffer

Copyright (C) 2010, David Beazley, http://www.dabeaz.com

Direct Writing

- write() and send() operations already know about array-like objects

```
>>> f = open("data.bin","wb")       # File in binary mode

>>> s = bytearray(b"Hello World\n") # Write a byte array
>>> f.write(s)
12

>>> import array
>>> a = array.array("i",[0,1,2,3,4,5])
>>> f.write(a)                        # Write an int array
24
```

Notice : An array of integers was written without any intermediate conversion

Copyright (C) 2010, David Beazley, http://www.dabeaz.com

Direct Reading

- You can read into an existing buffer/array using readinto() (and other *_into() variants)

```
>>> f = open("data.bin","rb")       # File in binary mode

>>> s = bytearray(12)               # Preallocate an array
>>> s
bytearray(b'\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00')
>>> f.readinto(s)                   # Read into it
12
>>> s
bytearray(b'Hello World\n')
>>>
```

- readinto() fills the supplied buffer and returns the actual number of bytes read

Copyright (C) 2010, David Beazley, http://www.dabeaz.com

Direct Reading

- Direct reading works with other arrays too

```
>>> a = array.array('i',[0])*10
>>> a
array('i', [0, 0, 0, 0, 0, 0, 0, 0, 0, 0])

>>> f.readinto(a)
24
>>> a
array('i', [0, 1, 2, 3, 4, 5, 0, 0, 0, 0])
>>>
```

- This is a feature that's meant to integrate well with extensions such as ctypes, numpy, etc.

Copyright (C) 2010, David Beazley, http://www.dabeaz.com

Direct Packing/Unpacking

- Direct access to memory buffers shows up in other library modules as well
- For example: struct

```
struct.pack_into(fmt, buffer, offset, ...)
struct.unpack_from(fmt, buffer, offset)
```

- Example use:

```
>>> a = bytearray(10)
>>> a
bytearray(b'\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00')
>>> struct.pack_into("HH",a,4,0xaaaa,0xbbbb)
>>> a
bytearray(b'\x00\x00\x00\x00\xaa\xaa\xbb\xbb\x00\x00')
>>>
```

Notice in-place packing of values

Copyright (C) 2010, David Beazley, http://www.dabeaz.com

Record Packing Revisited

- An example of in-place record packing

```
objs = [ ... ]         # List of tuples to pack
fmt  = "..."           # Format code

recsize = struct.calcsize(fmt)
msg = bytearray(4+len(objs)*recsize)

# First pack the number of objects
struct.pack_into("I",msg,0,len(objs))

# Incrementally pack each object
for n,x in enumerate(objs):
    struct.pack_into(fmt,msg,4+n*recsize,*x)

# Do something with the message
f.write(msg)
```

Copyright (C) 2010, David Beazley, http://www.dabeaz.com

memoryview Objects

- Direct I/O, in-place packing, and other features are tied to the buffer API (C) and memoryviews

```
>>> a = b"Hello World"
>>> v = memoryview(a)
>>> v
<memory at 0x45b210>
>>>
```

- A memory view directly exposes data as a buffer of bytes that can be used in low-level operations

Copyright (C) 2010, David Beazley, http://www.dabeaz.com

How Views Work

- A memory view is a memory overlay

```
>>> a = bytearray(10)
>>> a
bytearray(b'\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00')
>>> v = memoryview(a)
>>>
```

- If you read or modify the view, you're working with the same memory as the original object

```
>>> v[0] = b'A'
>>> v[-5:] = b'World'
>>> a
bytearray(b'A\x00\x00\x00\x00World')
>>>
```

In-place modifications

Copyright (C) 2010, David Beazley, http://www.dabeaz.com

How Views Work

- Memory views do not violate mutability

```
>>> s = b"Hello World"
>>> v = memoryview(s)
>>> v[0] = b'X'
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: cannot modify read-only memory
>>>
```

- That's good!

Copyright (C) 2010, David Beazley, http://www.dabeaz.com

How Views Work

- Memory views make zero-copy slices

```
>>> a = bytearray(10)
>>> a
bytearray(b'\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00')
>>> v = memoryview(a)
>>> left = v[:5]                # Make slices of the view
>>> right = v[5:]
>>> left[:] = b"Hello"          # Reassign view slices
>>> right[:] = b"World"
>>> a                           # Look at original object
bytearray(b'HelloWorld')
>>>
```

- This differs from how slices usually work
- Normally, slices make data copies

Copyright (C) 2010, David Beazley, http://www.dabeaz.com

Practical Use of Views

- memoryviews are not something that casual Python programmers should be using
- I would hate to maintain someone's code that was filled with tons of memoryview hacks
- However, memoryviews have great potential for programmers building libraries, frameworks, and low-level infrastructure (e.g., distributed computing, message passing, etc.)

Copyright (C) 2010, David Beazley, http://www.dabeaz.com

Practical Uses of Views

- Examples:
  - Incremental I/O processing
  - Message encoding/decoding
  - Integration with foreign software (C/C++)
- Big picture : It can be used to streamline the connections between different components by reducing memory copies

Copyright (C) 2010, David Beazley, http://www.dabeaz.com

Incremental Writing

- Create a massive bytearray (256MB)

```
>>> a = bytearray(range(256))*1000000
>>> len(a)
256000000
>>>
```

- Challenge : Blast the array through a socket
- Problem : If you know about sockets, you know that a single send() operation won't send 256MB.
- You've got to break it down into smaller sends

Copyright (C) 2010, David Beazley, http://www.dabeaz.com

Incremental Writing

- Here's an example of incremental transmission with memoryview slices

```
view = memoryview(a)
while view:
    nbytes = s.send(view)
    view = view[nbytes:]   # This is a zero-copy slice
```

- This sweeps over the bytearray, sending it in chunks, but never makes a memory copy

Copyright (C) 2010, David Beazley, http://www.dabeaz.com

Incremental Reading

- Suppose you wanted to incrementally read data into an existing byte array until it's filled

```
a = bytearray(size)
view = memoryview(a)
while view:
    nbytes = s.recv_into(view)
    view = view[nbytes:]
```

- If you know how much data is being received in advance, you can preallocate the array and incrementally fill it (again, no copies)

Copyright (C) 2010, David Beazley, http://www.dabeaz.com

Commentary

- Again, direct manipulation of memoryviews is something you probably want to avoid
- However, be on the lookout for functions such as read_into(), pack_into(), recv_into(), etc. in the standard library
- These make use of views and can offer I/O efficiency gains for programmers who know how to use them effectively

Copyright (C) 2010, David Beazley, http://www.dabeaz.com

Part 8

Porting to Python 3
(and final words)

Copyright (C) 2010, David Beazley, http://www.dabeaz.com

Big Picture

- I/O handling in Python 3 is so much more than minor changes to Python syntax
- It's a top-to-bottom redesign of the entire I/O stack that has new idioms and new features
- Question : If you're porting from Python 2, do you want to stick with Python 2 idioms or do you take full advantage of Python 3 features?

Copyright (C) 2010, David Beazley, http://www.dabeaz.com

Python 2 Backport

- Almost everything discussed in this tutorial has been back-ported to Python 2
- So, you can actually use most of the core Python 3 I/O idioms in your Python 2 code now
- Caveat : try to use the most recent version of Python 2 possible (e.g., Python 2.7)
- There is active development and bug fixes

Copyright (C) 2010, David Beazley, http://www.dabeaz.com

Porting Tips

- Make sure you very clearly separate bytes and unicode in your application
- Use the byte literal syntax : b'*bytes*'
- Use bytearray() for binary data handling
- Use new text formatting idioms (.format, etc.)

Copyright (C) 2010, David Beazley, http://www.dabeaz.com

Porting Tips

- When you're ready for it, switch to the new open() and print() functions

```
from __future__ import print_function
from io import open
```

- This switches to the new IO stack
- If you application still works correctly, you're well on your way to Python 3 compatibility

Copyright (C) 2010, David Beazley, http://www.dabeaz.com

Porting Tips

- Tests, tests, tests, tests, tests, tests...
- Don't even remotely consider the idea of Python 2 to Python 3 port without unit tests
- I/O handling is only part of the process
- You want tests for other issues (changed semantics of builtins, etc.)

Modernizing Python 2

- Even if Python 3 is not yet an option for other reasons, you can take advantage of its I/O handling idioms now
- I think there's a lot of neat new things
- Can benefit Python 2 programs in terms of more elegant programming, improved efficiency

That's All Folks!

- Hope you learned at least one new thing
- Please feel free to contact me

http://www.dabeaz.com

- Also, I teach Python classes (shameless plug)

Copyright (C) 2010, David Beazley, http://www.dabeaz.com