

# Mastering Python 3 I/O

(version 2.0)

David Beazley  
<http://www.dabeaz.com>

Presented at PyCon'2011  
Atlanta, Georgia

## This Tutorial

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

# Why I/O?

- Real programs interact with the world
  - They read and write files
  - They send and receive messages
- I/O is at the heart of almost everything that Python is about (scripting, data processing, gluing, frameworks, C extensions, etc.)
- Most tricky porting issues are I/O related

# The I/O Issue

- 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)

# The Plan

- We're going to take a detailed top-to-bottom tour of the Python 3 I/O system
  - Text handling, formatting, etc.
  - Binary data handling
  - The new I/O stack
  - System interfaces
  - Library design issues

# Prerequisites

- I assume that you are already somewhat 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 assumed

# Performance Disclosure

- There are some performance tests
- Execution environment for tests:
  - 2.66 GHZ 4-Core MacPro, 3GB memory
  - OS-X 10.6.4 (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

# Resources

- 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

# Part I

## Introducing Python 3

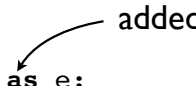
# Syntax Changes

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

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

- Exception handling syntax changes slightly

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



- Yes, your old code will break

# Many New Features

- Python 3 introduces many new features
- Composite string formatting

```
"{:10s} {:10d} {: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

# 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>  
>>>
```

- Python 3 prefers iterators/generators

# Library Reorganization

- The standard library has been cleaned up
- 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")
```

# 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
...
```

- 2to3 helps, but it's not foolproof (in fact, most of the time it doesn't quite work)

# 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

# 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 →
```



# 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?

# 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 %
```

Ah ha! Look at that!



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

# 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)

# Important Lessons

- A lot of things change in Python 3
- 2to3 only fixes really "obvious" things
- It does not fix I/O problems
- Why you should care : Real programs do I/O

# Part 2

## Working with Text

# Making Peace with Unicode

- In Python 3, all text is Unicode
- All strings are Unicode
- All text-based I/O is Unicode
- You can't ignore it or live in denial
- However, you don't have to be a Unicode guru

# 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

# 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"
- Denoted U+HHHH in polite conversation

ñ = U+00F1  
ε = U+03B5  
ς = U+0A87  
イテ = U+3304

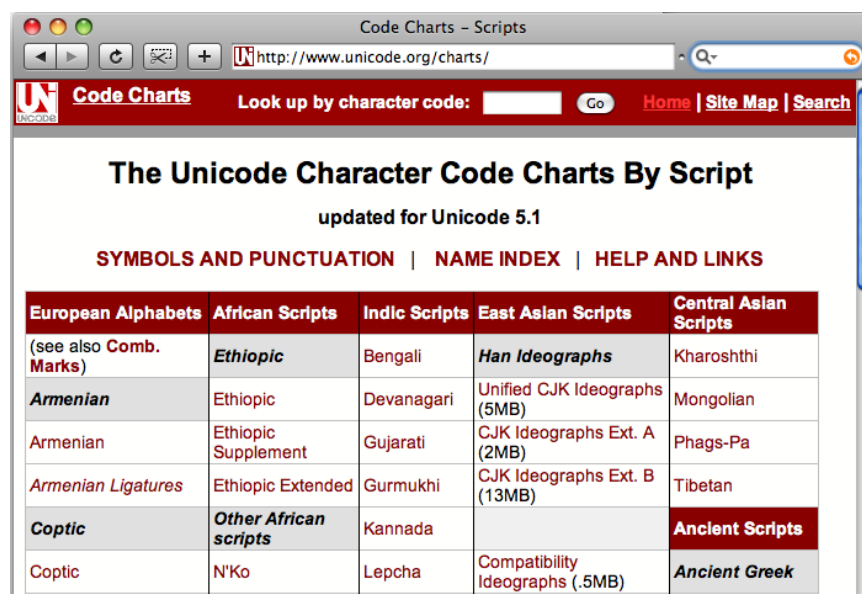
# Unicode Charts

- An issue : There are a lot of code points
- Largest 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.)

# Unicode Charts



The screenshot shows a web browser window displaying the 'Code Charts - Scripts' page. The page title is 'The Unicode Character Code Charts By Script', updated for Unicode 5.1. It features a navigation bar with 'SYMBOLS AND PUNCTUATION', 'NAME INDEX', and 'HELP AND LINKS'. Below the navigation bar is a table with five columns: European Alphabets, African Scripts, Indic Scripts, East Asian Scripts, and Central Asian Scripts. The table lists various scripts and their corresponding chart files, such as 'Ethiopic', 'Bengali', 'Han Ideographs', and 'Ancient Scripts'.

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

# Using Unicode Charts

- Consult to get code points for use in literals

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

	008	009	00A	00B	00C	00D	00E	00F
0	☒	☒	☒ NB SP	◦	À	Đ	à	ď
1	☒	☒ PU1	ı	±	Á	Ñ	á	ñ

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

- In practice : It doesn't come up that often

# Unicode Escapes

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

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

# A repr() Caution

- Python 3 source code is now Unicode
- Output of repr() is Unicode and doesn't use the escape codes (characters will be rendered)

```
>>> a = "Jalape\xfl0"  
>>> a  
'Jalapeño'
```

- Use ascii() to see the escape codes

```
>>> print(ascii(a))  
'Jalape\xfl0'  
>>>
```

# 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.) works fine, but is simply expanded

# A Caution

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

```
>>> s = "Jalape\x1f1o"
>>> t = "Jalapen\u0303o"
>>> s
'Jalapeño'
>>> t
'Jalapeño'
>>> s == t
False
>>> len(s), len(t)
(8, 9)
>>>
```

*'ñ' = 'n'+'~' (combining ~)*

- Many hairy bits
- However, that's also a different tutorial

# Unicode Representation

- Internally, Unicode character codes are just stored as arrays of C 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 which using the sys module

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

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



# 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
```

- See PEP 393 (possible change in future)

# Performance Impact

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

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

```
Python 2.7.1 (bytes)    : 11.5 s
```

```
Python 3.2 (UCS-2)     : 24.2 s
```

```
Python 3.2 (UCS-4)     : 47.5 s
```

- Slower because more bytes are moving

# Performance Impact

- Operations that process strings character by character often run at comparable speed
  - lower, upper, find, regexs, etc.

- Example:

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

Python 2.7.1 (bytes) : 37.9s (???)

Python 3.2 (UCS-2) : 6.9s

Python 3.2 (UCS-4) : 7.0s

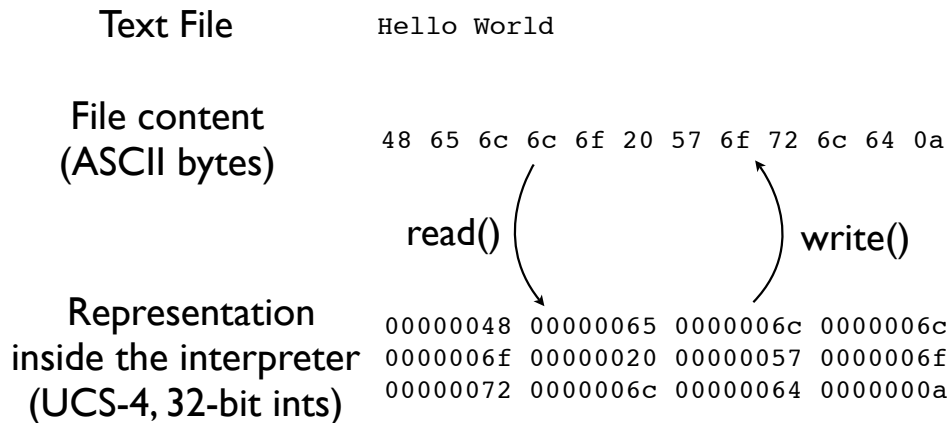
- The same number of iterations regardless of the size of each character

# Commentary

- Yes, unicode strings come at a cost
- Must study it if text-processing is a major component of your application
- Keep in mind--most programs do more than just string operations (overall performance impact might be far less than you think)

# Issue :Text Encoding

- The internal representation of characters is not the same as how characters are stored in files



# Issue :Text Encoding

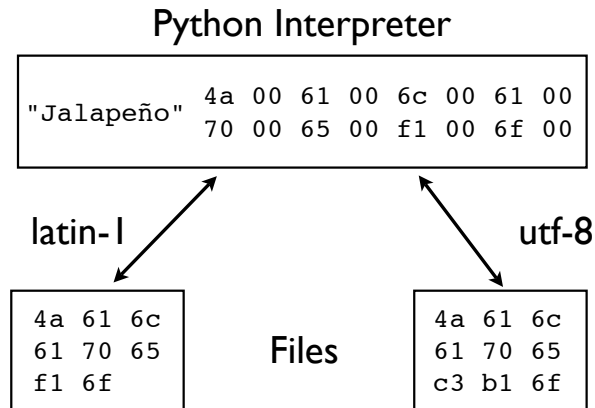
- There are also many possible char encodings for text (especially for non-ASCII chars)

	"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 :This is only related to how text is stored in files, not stored in memory

# Issue : Text Encoding

- Emphasize: text is always stored exactly the same way inside the Python interpreter



- It's only the encoding in files that varies

# 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).

# Reading/Writing Text

- Built-in `open()` function now 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.

# Encoding/Decoding Bytes

- Use `encode()` and `decode()` for byte strings

```
>>> s = "Jalapeño"
>>> data = s.encode('utf-8')
>>> data
b'Jalape\xc3\xb1o'

>>> data.encode('utf-8')
'Jalapeño'
>>>
```

- You'll need this for transmitting strings on network connections, passing to external systems, etc.

# Important Encodings

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

# 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)
>>>
```

# 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.)

# UTF-8 Encoding

- A multibyte variable-length encoding that can represent all Unicode characters

<u>Encoding</u>	<u>Description</u>
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:

$\tilde{n} = 0xf1 = \underline{11110001}$   
= 11000011 10110001 = 0xc3 0xb1 (UTF-8)

# UTF-8 Encoding

- Main feature of UTF-8 is that ASCII is embedded within it
- If you're not 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.)

# 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)



# Encoding Errors

- When working with Unicode, you might encounter encoding/decoding errors

```
>>> f = open('foo',encoding='ascii')
>>> data = f.read()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "/usr/local/lib/python3.2/encodings/
ascii.py", line 26, in decode
    return codecs.ascii_decode(input, self.errors)
[0]
UnicodeDecodeError: 'ascii' codec can't decode byte
0xc3 in position 6: ordinal not in range(128)
>>>
```

- This is almost always bad--must be fixed

# Fixing Encoding Errors

- Solution: Use the right encoding

```
>>> f = open('foo',encoding='utf-8')
>>> data = f.read()
>>>
```

- Bad Solution : Change the error handling

```
>>> f = open('foo',encoding='ascii',errors='ignore')
>>> data = f.read()
>>> data
'Jalapeo'
>>>
```

- My advice : Never use the errors argument without a really good reason. Do it right.

# Part 3

## Printing and Formatting

# New Printing

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

### Python 2

```
print x,y,z  
print x,y,z,  
print >>f,x,y,z
```

### Python 3

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

- `print()` has a few new tricks

# 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 were often requested (e.g., "how do I get rid of the space?")

# 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 tad bit slower)

# Object Formatting

- Here is Python 2 (%)

```
s = "%10.2f" % price
```

- Here is Python 3 (format)

```
s = format(price, "10.2f")
```

- This is part of a whole new formatting system

# Some History

- String formatting is one of the few features of Python 2 that can't be easily 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

# String Conversions

- Objects now have three string conversions

```
>>> x = 1/3
>>> x.__str__()
'0.3333333333333333'
>>> 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

## `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)
'0.3333333333333333'
>>> repr(x)
'0.3333333333333333'
>>>
```

# Format Codes (Builtins)

- For builtins, there are standard format codes

<u>Old Format</u>	<u>New Format</u>	<u>Description</u>
"%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

# 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%'
```

# 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

# 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'
>>>
```

# Fill Character

- Fill Character

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

- Examples:

```
>>> x = 42
>>> format(x, "08d")
'00000042'
>>> format(x, "032b")
'000000000000000000000000000000101010'
>>> format(x, " ^=32d")
'=====42====='
>>>
```

# 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'
>>>
```

- Alas, the use of the ',' isn't localized



# 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)`

# Composite Formatting

- String `.format()` method formats multiple values all at once (replacement for %)
- Some examples:

```
>>> "{name} has {n} messages".format(name="Dave",n=37)
'Dave has 37 messages'
```

```
>>> "{:10s} {:10d} {:10.2f}".format('ACME',50,91.1)
'ACME          50      91.10'
```

```
>>> "<{0}>{1}</{0}>".format('para','Hey there')
'<para>Hey there</para>'
>>>
```

# Composite Formatting

- `format()` method scans the string for formatting specifiers enclosed in `{ }` and expands each one
- Each `{ }` specifies what is being formatted as well as how it should be formatted
- Tricky bit : There are two aspects to it

# What to Format?

- You must specify arguments to `.format()`

- Positional:

```
"{0} has {1} messages".format("Dave", 37)
```

- Keyword:

```
"{name} has {n} messages".format(name="Dave", n=37)
```

- In order:

```
"{} has {} messages".format("Dave", 37)
```

# String Templates

- Template Strings

```
from string import Template

msg = Template("$name has $n messages")
print(msg.substitute(name="Dave",n=37))
```

- New String Formatting

```
msg = "{name} has {n} messages"
print(msg.format(name="Dave",n=37))
```

- Very similar

# Indexing/Attributes

- Cool thing :You can perform index lookups

```
record = {
    'name' : 'Dave',
    'n' : 37
}

'{r[name]} has {r[n]} messages'.format(r=record)
```

- Or attribute lookups with instances

```
record = Record('Dave',37)

'{r.name} has {r.n} messages'.format(r=record)
```

- Restriction: Can't have arbitrary expressions

# Specifying the Format

- Recall: There are three string format functions

```
str(s)
repr(s)
format(s, fmt)
```

- Each `{item}` can pick which it wants to use

```
{item}           # Replaced by str(item)
{item!r}         # Replaced by repr(item)
{item:fmt}       # Replaced by format(item, fmt)
```

# Format Examples


- More Examples:

```
>>> "{name:10s} {price:10.2f}".format(name='ACME', price=91.1)
'ACME           91.10'
```

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

```
>>> "{name!r}, {price}".format(name="ACME", price=91.1)
"'ACME', 91.1"
```

```
>>>
```

 note repr() output here

# Other Formatting Details

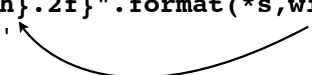
- { and } must be escaped if part of formatting
- Use '{{ for '{'
- Use '}}' for '}'
- Example:

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

# 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

# Formatting a Mapping

- Variation : `s.format_map(d)`

```
>>> record = {  
    'name' : 'Dave',  
    'n' : 37  
}  
>>> "{name} has {n} messages".format_map(record)  
'Dave has 37 messages'  
>>>
```

- This is a convenience function--allows names to come from a mapping without using `**`

# Commentary

- The new string formatting is very powerful
- The `%` operator will likely stay, but the new formatting adds more flexibility

# Part 4

## Binary Data Handling and Bytes

# 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

# 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("48656c6c66")
```

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

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

- However, this new bytes object is odd

# 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
```



# 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.

# Porting Note

- I have encountered a lot of minor problems with bytes in porting libraries

```
data = s.recv(1024)  
if data[0] == '+':  
    ...  
    ↓  
data = s.recv(1024)  
if data[0] == b'+': # ERROR!  
    ...  
    ↓  
data = s.recv(1024)  
if data[0] == 0x2b: # CORRECT  
    ...
```

# Porting Note

- Be careful with `ord()` (not needed)

```
data = s.recv(1024)
x = ord(data[0])
```

—————>

```
data = s.recv(1024)
x = data[0]
```

- Conversion of objects into bytes

```
>>> x = 7
>>> bytes(x)
b'\x00\x00\x00\x00\x00\x00\x00'

>>> str(x).encode('ascii')
b'7'
>>>
```

# 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')
>>>
```

# 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])
>>>
```

# 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.

# 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 mention that later...

# 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)

# 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.

# Passing Bytes as Strings

- Many library functions that work with "text" do not accept byte objects at all

```
>>> time.strptime(b"2010-02-17", "%Y-%m-%d")
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "/Users/beazley/Software/lib/python3.1/_strptime.py", line 461, in _strptime_time
    return _strptime(data_string, format)[0]
  File "/Users/beazley/Software/lib/python3.1/_strptime.py", line 301, in _strptime
    raise TypeError(msg.format(index, type(arg)))
TypeError: strptime() argument 0 must be str, not <class
'bytes'>
>>>
```

# 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

# How to Use Bytes

- Bytes are better suited for low-level I/O handling (message passing, distributed computing, embedded systems, etc.)
- I will show some examples that illustrate
- A complaint: documentation (online and books) is somewhat thin on explaining practical uses of bytes and bytearray objects

# Example : Reassembly

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

```
msg = b""
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)
```

# 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.

# 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
```

# 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.



# 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)
```

# Commentary

- I like 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

# Related : Buffers

- `bytearray()` is an example of a "buffer"
- `buffer` : A contiguous region of memory (e.g., allocated like a C/C++ array)

- There are many other examples:

```
a = array.array("i", [1,2,3,4,5])
b = numpy.array([1,2,3,4,5])
c = ctypes.ARRAY(ctypes.c_int,5)(1,2,3,4,5)
```

- Under the covers, they're all similar and often interchangeable with bytes (especially for I/O)

# Advanced : Memory Views

- `memoryview()`

```
>>> a = bytearray(b'Hello World')
>>> b = memoryview(a)
>>> b
<memory at 0x1007014d0>
>>> b[-5:] = b'There'
>>> a
bytearray(b'Hello There')
>>>
```

- It's essentially an overlay over a buffer
- It's very low-level and its use seems tricky
- I would probably avoid it

# Part 5

## The io module

# 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

# The open() function

- 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")
Notice how → <_io.TextIOWrapper name='foo.txt' encoding='UTF-8'>
you're getting a → >>> open("foo.txt", "rb")
different kind of → <_io.BufferedReader name='foo.txt'>
result here → >>> open("foo.txt", "rb", buffering=0)
               <_io.FileIO name='foo.txt' mode='rb'>
               >>>
```

# 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

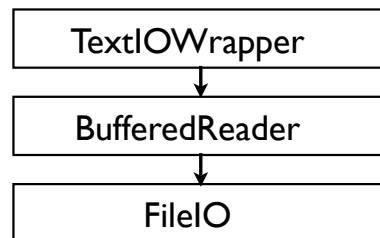
```
FileIO
BufferedReader
BufferedWriter
BufferedRWPair
BufferedRandom
TextIOWrapper
BytesIO
StringIO
```

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

# 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)

# 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()`

# 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)

# 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
- Systems programmers want this

# A Subtle Feature

- All files in Python 3 are opened in binary mode at the operating system level
- For Unix : Doesn't matter
- For Windows : It's subtle, but handling of newlines (and carriage returns) for text is now done by Python, not the operating system

# Commentary

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

# BufferedIO Objects

- The following classes implement buffered I/O

```
BufferedReader(f [, buffer_size])  
BufferedWriter(f [, buffer_size [, max_buffer_size]])  
BufferedReaderPair(f_read, f_write  
                  [, buffer_size [, max_buffer_size]])  
BufferedReaderRandom(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
```

# Buffering Behavior

- Buffering is controlled by two parameters (*buffer\_size* and *max\_buffer\_size*)
- *buffer\_size* is amount of data that can be stored before it's flushed to the I/O device
- *max\_buffer\_size* is the total amount of data that can be stored before blocking (default is twice *buffer\_size*).
- Allows more data to be accepted while previous I/O operation flush completes (useful for non-blocking I/O applications)



# 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

# File-Like Caution

- If you are making file-like objects, they may need the new `read1()` method

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

- Minimally alias it to `read()`
- If you leave it off, the program will crash if other code ever tries to access it

# 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
```

# Text Line Handling

- By default, files are opened in "universal" newline mode where all newlines are mapped to '\n'

```
>>> open("foo", "r").read()  
'Hello\nWorld\n'
```

- Use `newline=""` to return lines unmodified

```
>>> open("foo", "r", newline='').read()  
'Hello\r\nWorld\r\n'
```

- For writing, `os.linesep` is used as the newline unless otherwise specified with `newlines` parm

```
>>> f = open("foo", "w", newline='\r\n')  
>>> f.write('Hello\nWorld\n')
```

# TextIOWrapper and codecs

- Python 2 used the codecs module for unicode
- TextIOWrapper 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"): 3.8 sec
    pass
```

```
f = codecs.open("biglog.txt",encoding="utf-8") 53.3 sec
for line in f:
    pass
```

Note: both tests performed using Python-3.1.1

## 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

# open() Revisited

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

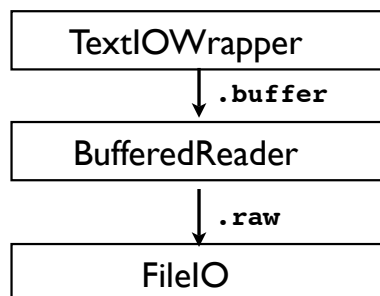
<u>mode</u>	<u>buffering</u>	<u>Result</u>
<i>any binary</i>	0	FileIO
"rb"	!= 0	BufferedReader
"wb", "ab"	!= 0	BufferedWriter
"rb+", "wb+", "ab+"	!= 0	BufferedReader
<i>any text</i>	!= 0	TextIOWrapper

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

# Unwinding the I/O Stack

- Sometimes you might need to unwind layers

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



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

# Unwinding Example

- Writing binary data on `sys.stdout`

```
>>> import sys
>>> sys.stdout.write(b"Hello World\n")
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: must be str, not bytes

>>> sys.stdout.buffer.write(b"Hello World\n")
Hello World
12
>>>
```

# Layering Caution

- The layering of I/O is buggy with file-like objects

```
>>> import io
>>> from urllib.request import urlopen
>>> u = io.TextIOWrapper(
    urlopen("http://www.python.org"),
    encoding='latin1')
>>> text = u.read()

>>> u = io.TextIOWrapper(
    urlopen("http://www.python.org"),
    encoding='latin1')
>>> line = u.readline()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'HTTPResponse' object has no
attribute 'readl'
```

- Will eventually sort itself out

# I/O Performance

- Question : How does new I/O perform?
- Will compare:
  - Python 2.7.1 built-in open()
  - Python 3.2 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

# I/O Performance

- Read a 100 Mbyte text file all at once

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

Python 2.7.1	:0.14s
Python 3.2 (UCS-2, UTF-8)	:0.90s
Python 3.2 (UCS-4, UTF-8)	:1.56s

Yes, you get  
overhead due to  
text decoding

- Read a 100 Mbyte binary file all at once

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

Python 2.7.1	:0.16s
Python 3.2 (binary)	:0.14s

(Not a significant  
difference)

- Note: tests conducted with warm disk cache

# I/O Performance

- Write a 100 Mbyte text file all at once

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

```
Python 2.7.1           : 1.73s  
Python 3.2 (UCS-2, UTF-8) : 1.85s  
Python 3.2 (UCS-4, UTF-8) : 1.85s
```

- Write a 100 Mbyte binary file all at once

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

```
Python 2.7.1           : 1.79s  
Python 3.2 (binary)    : 1.80s
```

- Note: tests conducted with warm disk cache

# I/O Performance

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

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

```
Python 2.7.1           : 0.25s  
Python 3.2 (UCS-2, UTF-8) : 0.57s  
Python 3.2 (UCS-4, UTF-8) : 0.82s
```

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

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

```
Python 2.7.1           : 0.25s  
Python 3.2 (binary)    : 0.29s
```

# I/O Performance

- Write 730000 lines log data (text)

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

Python 2.7.1	: 1.2s	
Python 3.2 (UCS-2, UTF-8)	: 1.2s	(10 sample averages, not an
Python 3.2 (UCS-4, UTF-8)	: 1.2s	observation difference)

- Write 730000 lines of log data (binary)

```
open("biglog.txt", "wb").writelines(binlines)
```

Python 2.7.1	: 1.2s
Python 3.2 (binary)	: 1.2s

# 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



# 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)

# 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

# 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

## 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 2.71           : 1.22s
Python 3.2 (UCS-2)   : 1.73s
Python 3.2 (UCS-4)   : 2.00s
```

# 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])
error_404_urls = {n.decode('latin-1')
                  for n in error_404_urls }

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

Unicode conversion here



Python 3.2 (UCS-2) : 1.29s (down from 1.73s)

Python 3.2 (UCS-4) : 1.28s (down from 2.00s)

## Part 6

### System Interfaces

# 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

## A 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

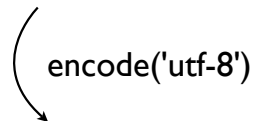
# Question

- 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

# 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

# Arguments & Environ

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

Python 3:

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

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

# 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

# Example :A Bad Filename

- Start Python 2 on Linux and create a file using the `open()` function like this:

```
>>> f = open("jalape\xfl0.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?

# Example :A Bad Filename

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

```
>>> f = open("jalape\xfl0.txt")
Traceback (most recent call last):
...
IOError: [Errno 2] No such file or directory: 'jalapeño.txt'
>>>
```

- This is caused by an encoding mismatch

```
"jalape\xfl0.txt"
      ↓ UTF-8
b"jalape\xc3\xb10.txt"
```

```
      ↓ open()
Fails!
```

It fails because this is  
the actual filename

```
b"jalape\xfl0.txt"
```

# 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!

# Names as Bytes

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

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

Notice bytes

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



# 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

# Surrogate Encoding

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

- Example:

```
b"jalape\xfflo.txt"  
      ↓ surrogate encoding  
"jalape\udcfflo.txt"
```

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

# 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 through a system interface

# Surrogate Encoding

- Question : Does this break part of Unicode?
- Answer : Unsure
- It 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

# Caution : Printing

- Non-decodable bytes will break print()

```
>>> files = glob.glob("*.txt")
>>> files
[ 'jalape\udcflo.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

# Implementation

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

- Example:

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

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

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

# 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?)

## Part 7

### Library Design Issues

# Text, Bytes, and Libraries

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

## Example : Socket Sends

- Here's a "broken" function

```
def send_response(s, code, msg):  
    s.sendall("HTTP/1.0 %s %s\r\n" % (code, msg))  
  
send_response(s, "200", "OK")
```

- Reason it's broken: sockets only work with binary I/O (bytes, bytearray, etc.)
- Passing text just isn't allowed

# 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" % (code,msg)  
    s.sendall(resp.encode('ascii'))  
  
send_response(s,"200","OK")
```

- Rules of thumb:
  - All outgoing text must be encoded
  - All incoming text must be decoded

# Discussion

- Where do you perform the encoding?
- At the point of data transmission?
- Or do you make users specify bytes elsewhere?

```
def send_response(s,code,msg):  
    resp = b"HTTP/1.0 " + code + b" " + msg + b"\r\n"  
    s.sendall(resp)  
  
send_response(s,b"200",b"OK")
```

# Discussion

- Do you write code that accepts str/bytes?

```
def send_response(s,code,msg):
    if isinstance(code,str):
        code = code.encode('ascii')
    if isinstance(msg,str):
        msg = msg.encode('ascii')
    resp = b"HTTP/1.0 " + code + b" " + msg + b"\r\n"
    s.sendall(resp)
```

```
send_response(s,b"200",b"OK")      # Works
send_response(s,"200","OK")       # Also Works
```

- If you do this, does it violate Python 3's strict separation of bytes/unicode?
- I have no answer

# More Discussion

- What about C extensions?

```
void send_response(int fd, const char *msg) {
    ...
}
```

- Is char \* bytes?
- Is char \* text? (Unicode)
- Is it both with implicit encoding?

# Muddled Text

- Is this the right behavior? (notice result)

```
>>> data = b'Hello World'  
>>> import base64  
>>> base64.b64encode(data)  
b'SGVsbG8gV29ybGQ='  
>>>
```

should this be bytes?

- It gets tricky once you start embedding all of these things into other data

## Part 8

Porting to Python 3  
(and final words)



# 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?

# 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

# 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.)

# Porting Tips

- Consider using a mockup of the new bytes type for differences in indexing/iteration

```
class _b(str):
    def __getitem__(self, index):
        return ord(str.__getitem__(self, index))
```

- Example:

```
>>> s = _b("Hello World")
>>> s[0]
72
>>> for c in s: print c,
...
72 101 108 108 111 32 87 111 114 108 100
```

- Put it around all use of bytes and make sure your code still works afterwards (in Python 2)

# Porting Tips

- StringIO has been split into two classes

```
from io import StringIO, BytesIO

f = StringIO(text) # StringIO for text only (unicode)
g = BytesIO(data) # BytesIO for bytes only>>>
```

- Be very careful with the use of StringIO in unit tests (where I have encountered most problems)

# 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 your application still works correctly, you're well on your way to Python 3 compatibility

# 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)