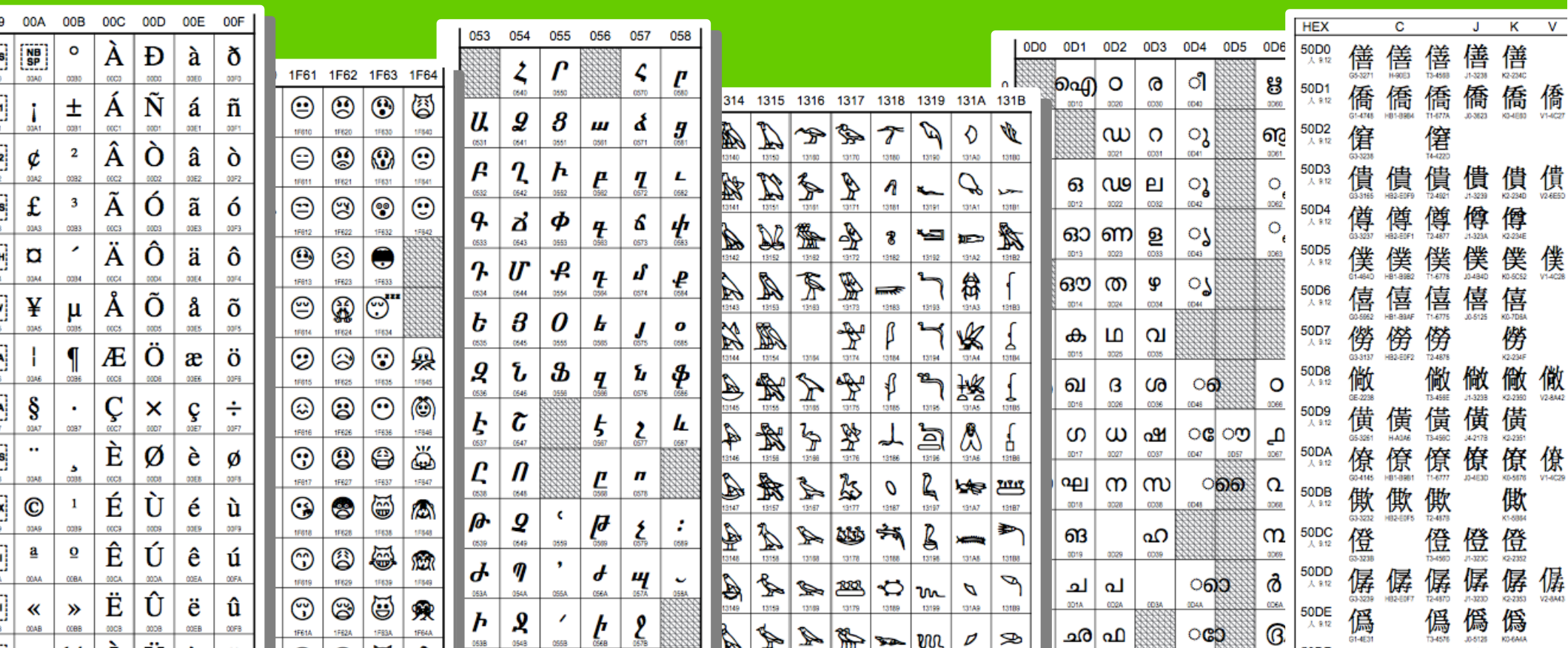
Unicode solutions in Python 2 and 3

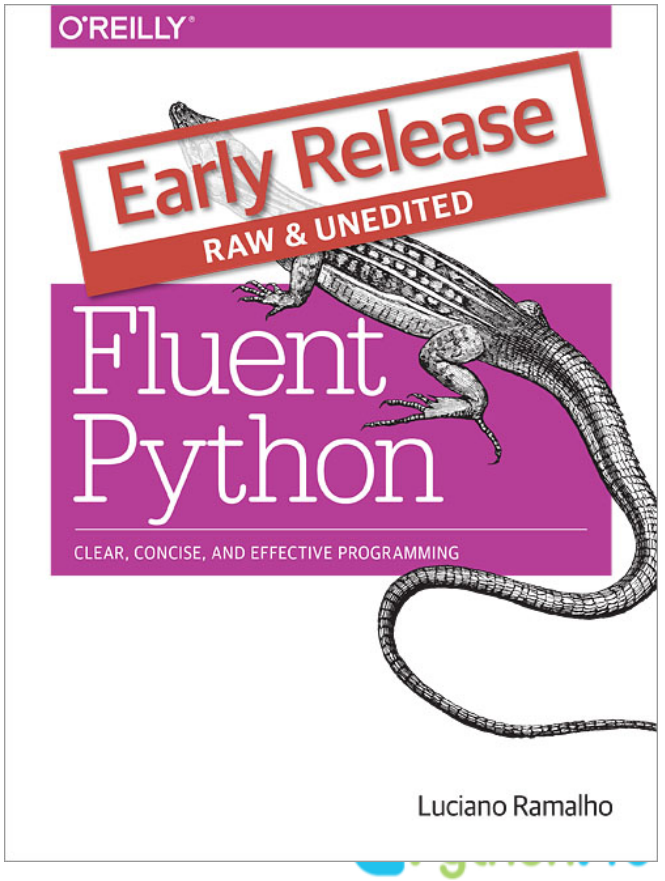


Unicode solutions in Python 2 and 3

About me: Luciano Ramalho

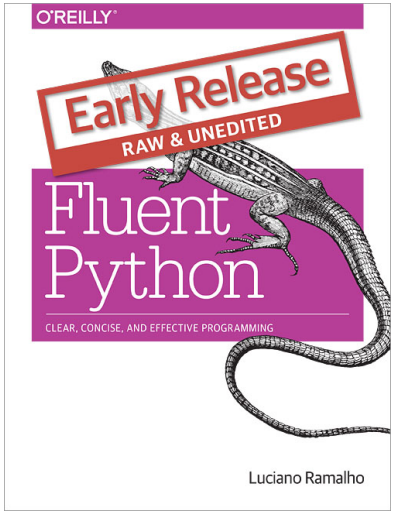
- Programming in Python since 1998
- Focus on content management (i.e. text wrangling)
- Teaching Python since 1999
- Speaker at PyCon US, OSCON, FISL, PythonBrasil, RuPy, QCon...
- Author of ***Fluent Python***
- Twitter: **@ramalhoorg**
- Native language: Português
  - "ação"

4 non-ASCII characters here



Resources

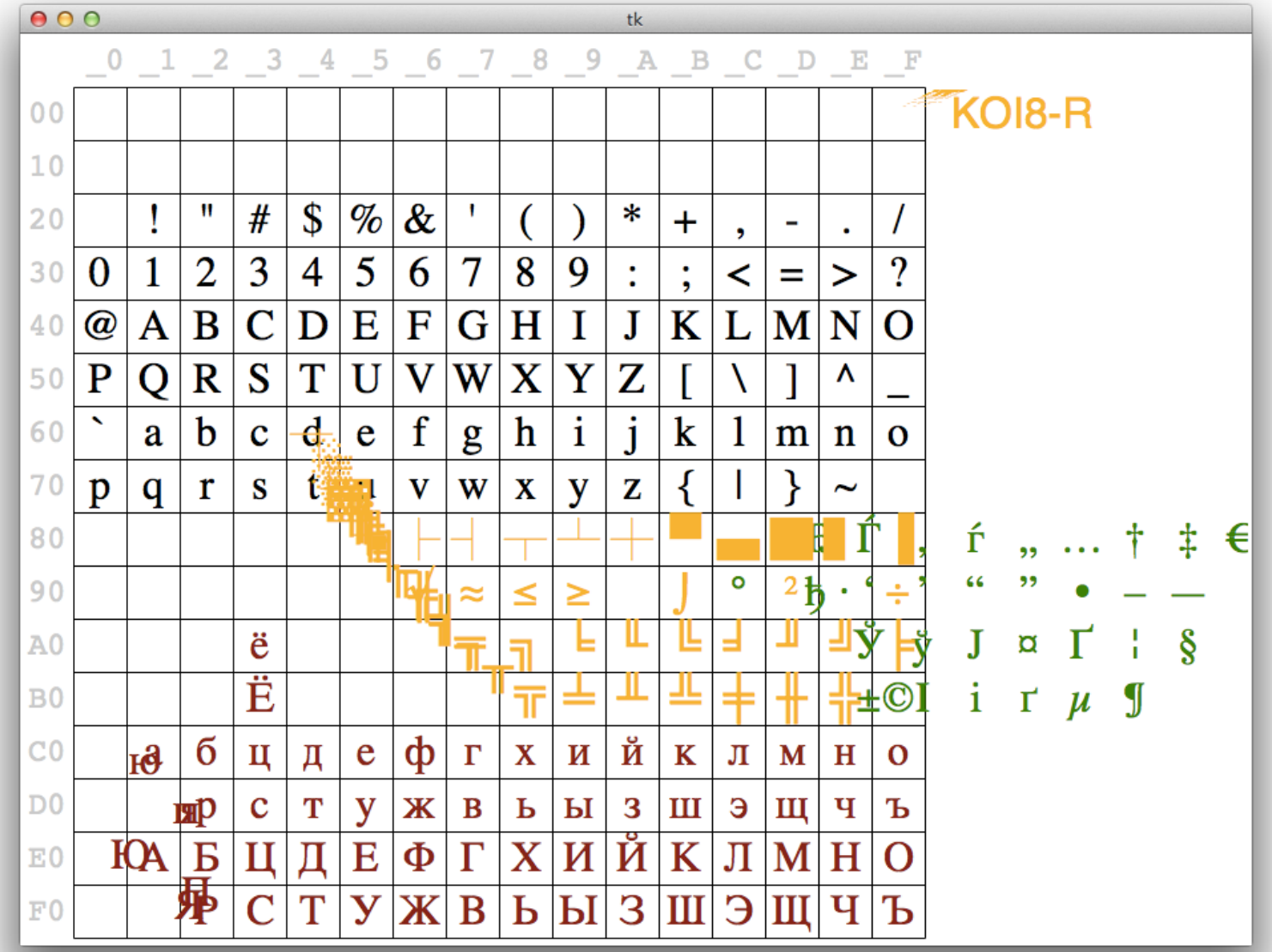
- All code, slides and images used in this talk:
  - https://github.com/fluentpython/unicode-solutions
- **Fluent Python**
  - http://shop.oreilly.com/product/0636920032519.do
  - Relevant content and examples:
    - Chapter 4: *Text versus Bytes*
      - all 39 pages
    - Chapter 18: *Concurrency with asyncio*
      - the *charfinder* examples



PythonPro

A bite of theory

The single-byte codepage ballet



Video: https://www.youtube.com/watch?v=J4qioAacrYo

Source code: http://bit.ly/1Oqt0MZ

PythonPro

Why Unicode

- Too many incompatible byte encodings
- Separate concepts:
  - character identity: one **code point** for each abstract character
    - `U+0041 → LATIN CAPITAL LETTER A`
    - `U+096C → DEVANAGARI DIGIT SIX`
  - binary representation: multiple **encodings**
    - `U+0041 → 0x41` | `0x41 0x00`
    - `U+096C → 0xE0 0xA5 0xAC` | `0x6C 0x09`

A sample of encodings

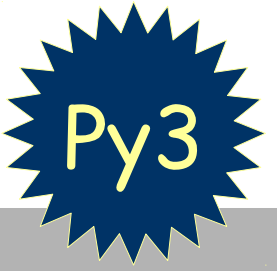
char.	code point	ascii	latin1	cp1252	cp437	gb2312	utf-8	utf-16le
A	U+0041	41	41	41	41	41	41	41 00
¿	U+00BF	*	BF	BF	A8	*	C2 BF	BF 00
Ã	U+00C3	*	C3	C3	*	*	C3 83	C3 00
á	U+00E1	*	E1	E1	A0	A8 A2	C3 A1	E1 00
Ω	U+03A9	*	*	*	EA	A6 B8	CE A9	A9 03
ۿ	U+06BF	*	*	*	*	*	DA BF	BF 06
“	U+201C	*	*	93	*	A1 B0	E2 80 9C	1C 20
€	U+20AC	*	*	80	*	*	E2 82 AC	AC 20
┌	U+250C	*	*	*	DA	A9 B0	E2 94 8C	0C 25
气	U+6C14	*	*	*	*	C6 F8	E6 B0 94	14 6C
氣	U+6C23	*	*	*	*	*	E6 B0 A3	23 6C
𝄞	U+1D11E	*	*	*	*	*	F0 9D 84 9E	34 D8 1E DD

Figure 4-1 of ***Fluent Python***

Byte and text solutions

	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
2	BPH 0082	PU2 0092	¢ 00A2	² 00B2	Â 00C2	Ò 00D2	â 00E2	ò 00F2
3	NBH 0083	STS 0093	£ 00A3	³ 00B3	Ã 00C3	Ó 00D3	ã 00E3	ó 00F3
4	IND 0084	CCH 0094	¤ 00A4	´ 00B4	Ä 00C4	Ô 00D4	ä 00E4	ô 00F4
5	NEL 0085	MW 0095	¥ 00A5	µ 00B5	Å 00C5	Õ 00D5	å 00E5	õ 00F5
6	SSA 0086	SPA 0096	¦ 00A6	¶ 00B6	Æ 00C6	Ö 00D6	æ 00E6	ö 00F6
7	ESA 0087	EPA 0097	§ 00A7	· 00B7	Ç 00C7	× 00D7	ç 00E7	÷ 00F7
8	HTS 0088	SOS 0098	¨ 00A8	¸ 00B8	È 00C8	Ø 00D8	è 00E8	ø 00F8
9	HTJ 0089	XXX 0099	© 00A9	¹ 00B9	É 00C9	Ù 00D9	é 00E9	ù 00F9
A	VTS 008A	SCI 009A	ª 00AA	º 00BA	Ê 00CA	Ú 00DA	ê 00EA	ú 00FA
B	PLD 008B	CSI 009B	« 00AB	» 00BB	Ë 00CB	Û 00DB	ë 00EB	û 00FB
C	PLU 008C	ST 009C	¬ 00AC	¼ 00BC	Ì 00CC	Ü 00DC	ì 00EC	ü 00FC
D	RI 008D	OSC 009D	SHY 00AD	½ 00BD	Í 00CD	Ý 00DD	í 00ED	ý 00FD
E	SS2 008E	PM 009E	® 00AE	¾ 00BE	Î 00CE	Þ 00DE	î 00EE	þ 00FE
F	SS3 008F	APC 009F	¯ 00AF	¿ 00BF	Ï 00CF	ß 00DF	ï 00EF	ÿ 00FF

Data types for text or bytes

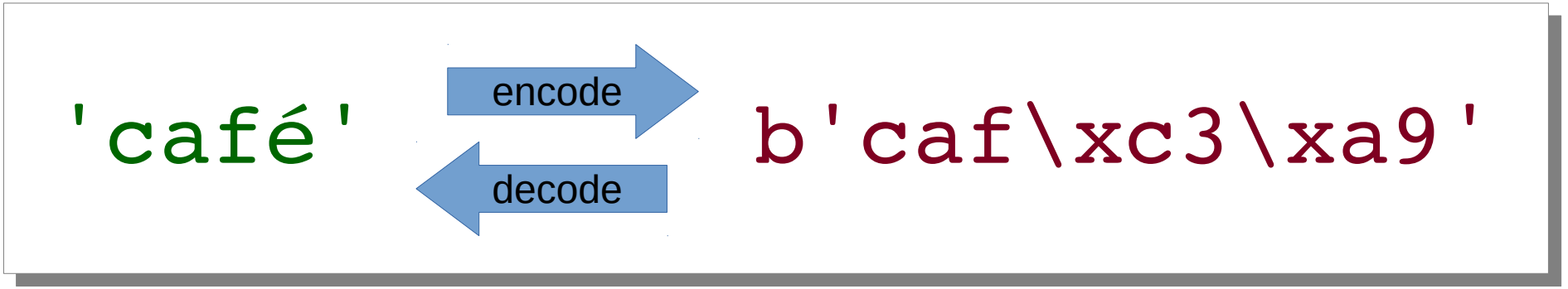


	Python 2.7	Python 3.4
Human text	**unicode** `u'café', u'caf\xe9'`	**str** `'café', u'café', 'caf\xe9'`
(immutable) **Bytes**	**str** `'café', 'caf\xe9', b'café'`	**bytes** `b'caf\xc3\xa9'`
(mutable) **Bytes**	**bytearray** `bytearray(b'caf\xc3\xa9')`	**bytearray** `bytearray(b'caf\xc3\xa9')`

PythonPro

.encode() v. .decode()

- “Humans use text. Computers speak bytes.”
  - Esther Nam and Travis Fischer in *Character encoding and Unicode in Python (Pycon US 2014)*



- Use **.encode()** to convert **human** text to **bytes**
- Use **.decode()** to convert **bytes** to **human** text

2.7 gotcha: methods .encode() and .decode() exist in both str and unicode types!

Text v. bytes


```
>>> text = 'café'
>>> type(text)
<class 'str'>
>>> len(text)
4
>>> list(text)
['c', 'a', 'f', 'é']
>>> print(text)
café
>>> octets = text.encode('utf8')
>>> type(octets)
<class 'bytes'>
>>> octets
b'caf\xc3\xa9'
>>> len(octets)
5
>>> list(octets)
[99, 97, 102, 195, 169]
>>> print(octets)
b'caf\xc3\xa9'
>>> octets.decode('utf8')
'café'
>>> str is bytes
False
```

- Items in unicode text are characters
- Items in byte sequences are bytes
  - integers 0...255
  - shown as ASCII sequences with a **b''** prefix for convenience

Text v. bytes

Py3

```
>>> text = 'café'
>>> type(text)
<class 'str'>
>>> len(text)
4
>>> list(text)
['c', 'a', 'f', 'é']
>>> print(text)
café
>>> octets = text.encode('utf8')
>>> type(octets)
<class 'bytes'>
>>> octets
b'caf\xc3\xa9'
>>> len(octets)
5
>>> list(octets)
[99, 97, 102, 195, 169]
>>> print(octets)
b'caf\xc3\xa9'
>>> octets.decode('utf8')
'café'
>>> str is bytes
False
```

Py2

```
>>> text = u'café'
>>> type(text)
<type 'unicode'>
>>> len(text)
4
>>> list(text)
[u'c', u'a', u'f', u'\xe9']
>>> print(text)
café
>>> octets = text.encode('utf8')
>>> type(octets)
<type 'str'>
>>> octets
'caf\xc3\xa9'
>>> len(octets)
5
>>> list(octets)
['c', 'a', 'f', '\xc3', '\xa9']
>>> print(octets)
café
>>> octets.decode('utf8')
u'caf\xe9'
>>> str is bytes
True
```

Bytes and bytearray


```
>>> octets = bytes('café', encoding='utf_8')
>>> octets
b'caf\xc3\xa9'
>>> octets[0]
99
>>> octets[:1]
b'c'
>>> octet_arr = bytearray(octets)
>>> octet_arr
bytearray(b'caf\xc3\xa9')
>>> octet_arr[-1:]
bytearray(b'\xa9')
```

Bytes and bytearray

Py3

```
>>> octets = bytes('café', encoding='utf_8')
>>> octets
b'caf\xc3\xa9'
>>> octets[0]
99
>>> octets[:1]
b'c'
>>> octet_arr = bytearray(octets)
>>> octet_arr
bytearray(b'caf\xc3\xa9')
>>> octet_arr[-1:]
bytearray(b'\xa9')
```

Py2

```
>>> octets = bytes('café')
>>> octets
'caf\xc3\xa9'
>>> octets[0]
'c'
>>> octets[:1]
'c'
>>> octet_arr = bytearray(octets)
>>> octet_arr
bytearray(b'caf\xc3\xa9')
>>> octet_arr[-1:]
bytearray(b'\xa9')
```

PythonPro

Common codecs

- codec = encoder/decoder table or algorithm

```
>>> for codec in ['latin_1', 'utf_8', 'utf_16']:
...     print('{:8} {!r}'.format(codec, u'El Niño'.encode(codec)))
...
latin_1  b'El Ni\xf1o'
utf_8    b'El Ni\xc3\xb1o'
utf_16   b'\xff\xfeE\x00l\x00 \x00N\x00i\x00\xf1\x00o\x00'
```

PythonPro

Common codecs

- codec = encoder/decoder table or algorithm

```
>>> for codec in ['latin_1', 'utf_8', 'utf_16']:
...     print('{:8} {!r}'.format(codec, u'El Niño'.encode(codec)))
...
latin_1  b'El Ni\xf1o'
utf_8    b'El Ni\xc3\xb1o'
utf_16   b'\xff\xfeE\x00l\x00 \x00N\x00i\x00\xf1\x00o\x00'
```


```
>>> for codec in ['latin_1', 'utf_8', 'utf_16']:
...     print '{:8} {!r}'.format(codec, u'El Niño'.encode(codec))
...
latin_1  'El Ni\xf1o'
utf_8    'El Ni\xc3\xb1o'
utf_16   '\xff\xfeE\x00l\x00 \x00N\x00i\x00\xf1\x00o\x00'
```

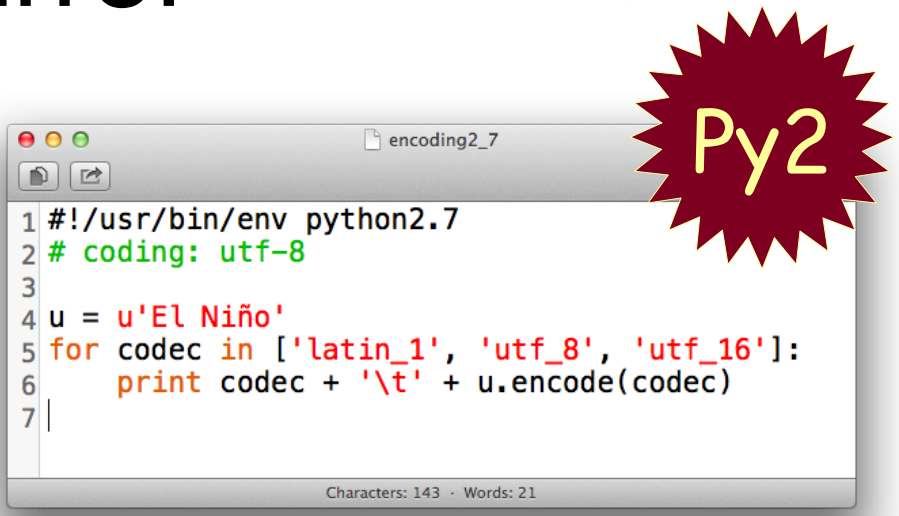
PythonPro

Coping with Unicode Errors

- **SyntaxError**
  - A **.py** file has source code in an unexpected encoding
- **UnicodeDecodeError**
  - A binary sequence is contains bytes that are not valid in the expected encoding
- **UnicodeEncodeError**
  - A Unicode string contains codepoints that have no representation in the desired encoding

Coping with SyntaxError

- A **.py** file uses an unexpected encoding
  - The source file encoding is not the default, and no `# coding` comment was found.
  - The source file encoding is not the one declared in the `# coding` comment
- Default source encoding:
  - Python 2.7 → ASCII
  - Python 3.x → UTF-8

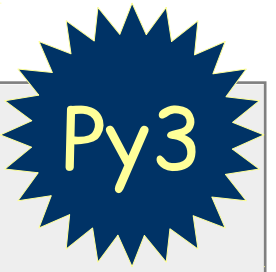


2.7 gotcha:
default source
encoding is ASCII

PythonPro

UnicodeEncodeError

- A character in the Unicode text cannot be represented in the target byte encoding
  - happens with legacy encodings that cover only a small subset of Unicode




```
>>> city = 'São Paulo'
>>> city.encode('utf_8')
b'S\xc3\xa3o Paulo'
>>> city.encode('utf_16')
b'\xff\xfeS\x00\xe3\x00o\x00 \x00P\x00a\x00u\x00l\x00o\x00'
>>> city.encode('iso8859_1')
b'S\xe3o Paulo'
>>> city.encode('cp437')
Traceback (most recent call last):
  ...
UnicodeEncodeError: 'charmap' codec can't encode character '\xe3' in position 1: character maps to <undefined>
>>> city.encode('cp437', errors='ignore')
b'So Paulo'
>>> city.encode('cp437', errors='replace')
b'S?o Paulo'
>>> city.encode('cp437', errors='xmlcharrefreplace')
b'S&#227;o Paulo'
```

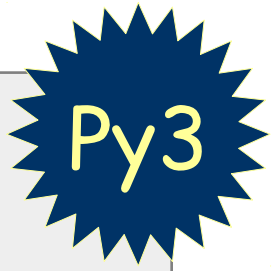
UnicodeEncodeError

- A character in the Unicode text cannot be represented in the target byte encoding
  - happens with legacy encodings that cover only a small subset of Unicode


```
>>> city = u'São Paulo'
>>> city.encode('utf_8')
'S\xc3\xa3o Paulo'
>>> city.encode('utf_16')
'\xff\xfeS\x00\xe3\x00o\x00 \x00P\x00a\x00u\x00l\x00o\x00'
>>> city.encode('iso8859_1')
'S\xe3o Paulo'
>>> city.encode('cp437')
Traceback (most recent call last):
  ...
UnicodeEncodeError: 'charmap' codec can't encode character u'\xe3' in position 1: character maps to <undefined>
>>> city.encode('cp437', errors='ignore')
'So Paulo'
>>> city.encode('cp437', errors='replace')
'S?o Paulo'
>>> city.encode('cp437', errors='xmlcharrefreplace')
'S&#227;o Paulo'
```

UnicodeDecodeError

- Invalid byte in the source encoding
  - more common with the UTF encodings




```
>>> octets = b'Montr\xe9al'
>>> octets.decode('cp1252')
'Montréal'
>>> octets.decode('iso8859_7')
'Montrιal'
>>> octets.decode('koi8_r')
'MontrИal'
>>> octets.decode('utf_8')
Traceback (most recent call last):
  ...
UnicodeDecodeError: 'utf-8' codec can't decode byte 0xe9 in position 5:
invalid continuation byte
>>> octets.decode('utf_8', errors='replace')
'Montr�al'
```

UnicodeDecodeError


```
>>> octets = b'Montr\xe9al'
>>> octets.decode('cp1252')
u'Montr\xe9al'
>>> print _
Montréal
>>> octets.decode('iso8859_7') == u'Montrıal'
True
>>> print octets.decode('iso8859_7')
Montrıal
>>> octets.decode('koi8_r') == u'MontrИal'
True
>>> print octets.decode('koi8_r')
MontrИal
>>> octets.decode('utf_8')
Traceback (most recent call last):
  ...
UnicodeDecodeError: 'utf8' codec can't decode byte 0xe9 in position 5:
invalid continuation byte
>>> octets.decode('utf_8', errors='replace') == u'Montr\ufffdal'
True
>>> print octets.decode('utf_8', errors='replace')
Montr\ufffdal
```

Best practice to avoid errors

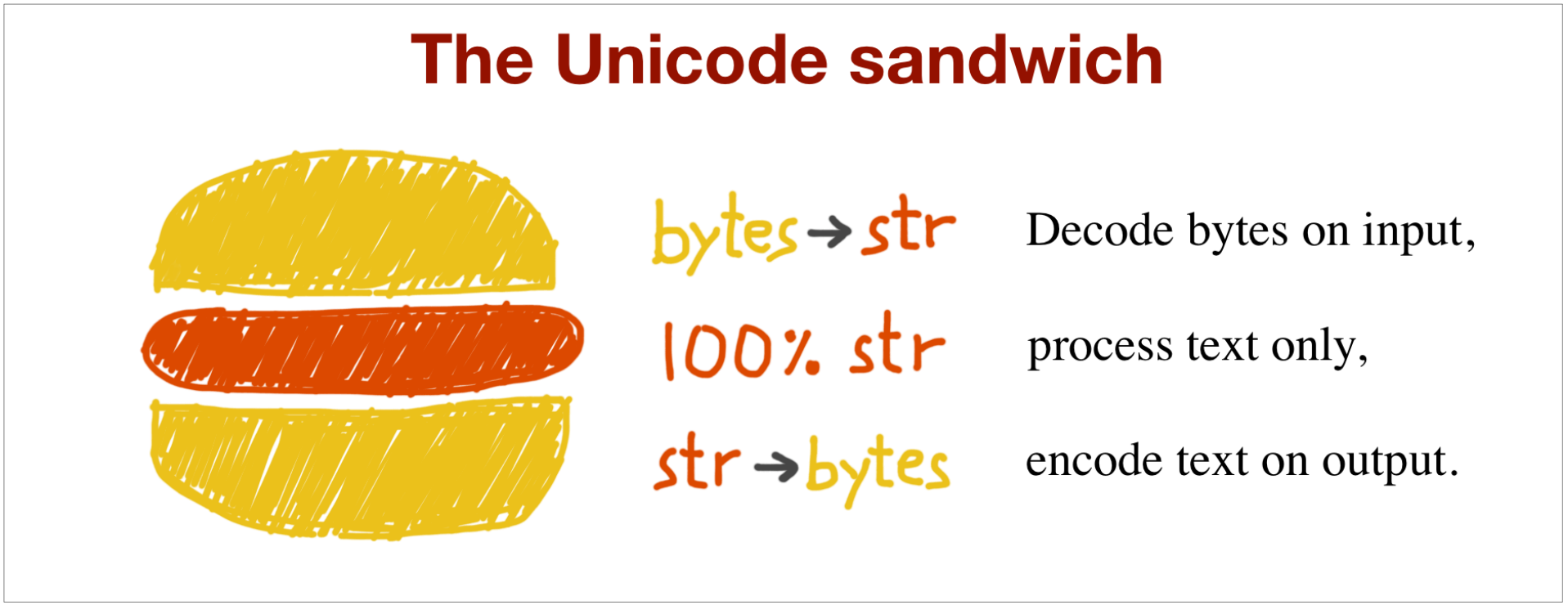


*Figure 4-2 of **Fluent Python**, after Ned Batchelder's **Pragmatic Unicode** talk: http://nedbatchelder.com/text/unipain.html*

PythonPro

How to implement the sandwich

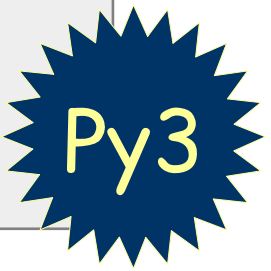
- Always specify encoding when opening text files, so you send and receive text, and not bytes
  - in Python 2.7, remember to use `io.open()`

2.7 gotcha:
no way to specify encoding
with built-in open(...).
Must use io.open(...).

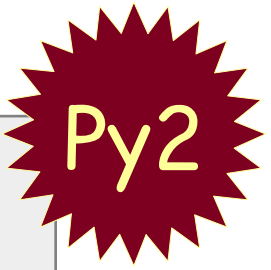
Text I/O

- **open()** built-in is Unicode-aware in Python 3
  - text mode default accepts **encoding** argument
  - **.write()** method only accepts Unicode text
  - **.read()** method returns Unicode text

```
>>> with open('cafe.txt', 'w', encoding='utf_8') as fp:
...     write_count = fp.write('café')
...
>>> write_count
4
>>> with open('cafe.txt', 'r', encoding='utf_8') as fp:
...     text = fp.read()
...
>>> text
'café'
```



Bytes or text I/O




```
>>> with open('cafe.txt', 'w') as fp:
...     fp.write('café')
...
>>> with open('cafe.txt', 'r') as fp:
...     octets = fp.read()
...
>>> octets
'caf\xc3\xa9'
```

- **open()** built-in only supports bytes in Python 2
  - even in “text mode” (deals with CR+LF...)
  - no **encoding** argument accepted
  - **.write()** implicitly converts **unicode** to **str** using ASCII codec
    - remember: `'café'` is actually `b'caf\xc3\xa9'`
  - **.read()** method always returns bytes

Bytes or text I/O


```
>>> with open('cafe.txt', 'w') as fp:
...     fp.write('café')
...
>>> with open('cafe.txt', 'r') as fp:
...     octets = fp.read()
...
>>> octets
'caf\xc3\xa9'
```

```
>>> import io
>>> with io.open('cafe.txt', 'w', encoding='utf_8') as fp:
...     write_count = fp.write(u'café')
...
>>> write_count
4L
>>> with io.open('cafe.txt', 'r', encoding='utf_8') as fp:
...     text = fp.read()
...
>>> text
u'caf\xe9'
```

- **io.open()** is the Unicode-aware **open()** from Python 3 backported to Python 2.6+

Bytes or text I/O


```
>>> with open('cafe.txt', 'w') as fp:
...     fp.write('café')
...
>>> with open('cafe.txt', 'r') as fp:
...     octets = fp.read()
...
>>> octets
'caf\xc3\xa9'
```

```
>>> import io
>>> with io.open('cafe.txt', 'w', encoding='utf_8') as fp:
...     write_count = fp.write(u'café')
...
>>> write_count
4L
>>> with io.open('cafe.txt', 'r', encoding='utf_8') as fp:
...     text = fp.read()
...
>>> text
u'caf\xe9'
```

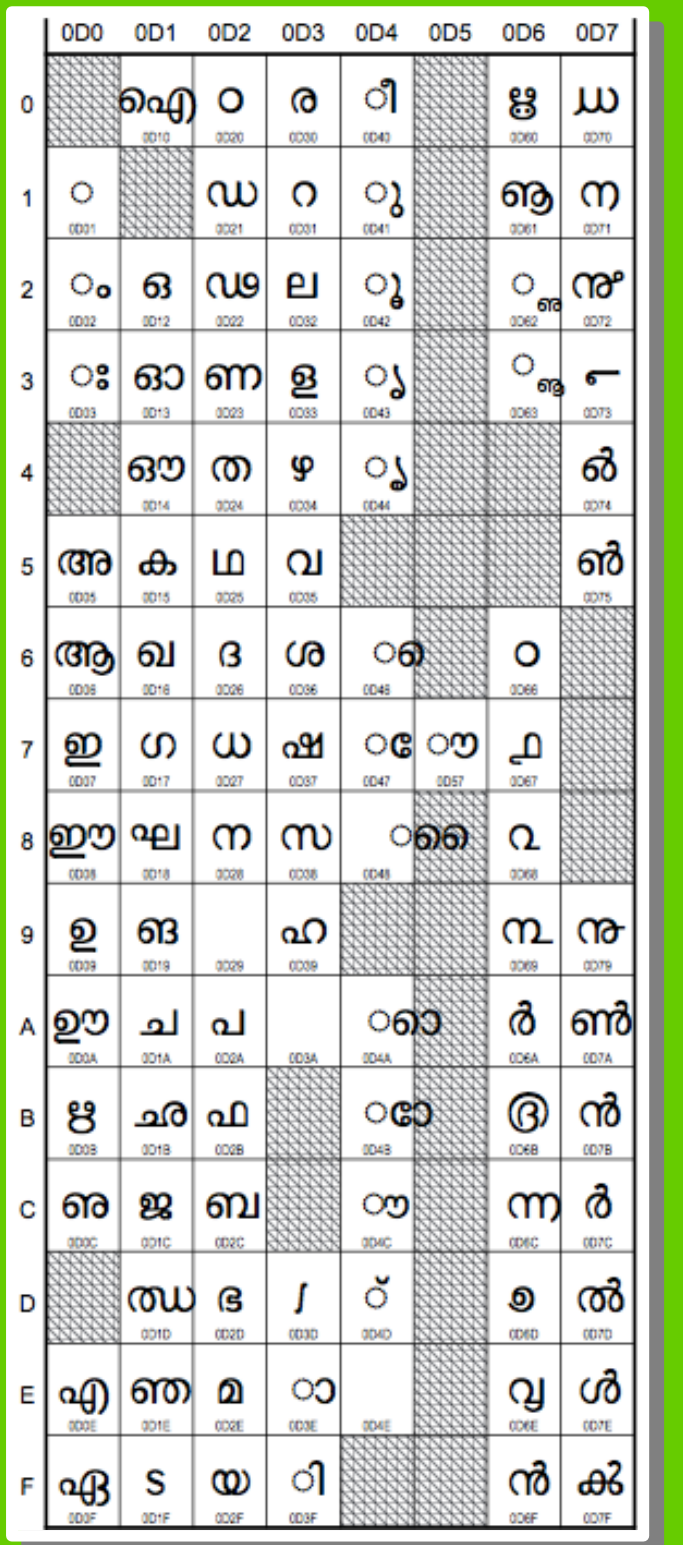
```
>>> with io.open('cafe.txt', 'wb') as fp:
...     write_count = fp.write('café')
...
>>> write_count
5L
>>> with io.open('cafe.txt', 'rb') as fp:
...     octets = fp.read()
...
>>> octets
'caf\xc3\xa9'
```

- **io.open()** also handles bytes
  - mode 'b'

FAQ: How to find out the encoding of a file?

- Some files have an encoding header
  - HTML, XML, *some* database dumps
- Otherwise, you must be told. Ask!
- If you can't ask, try the Chardet package
  - not 100% safe, but pretty smart
  - uses statistics and heuristics
  - includes a `chardetect` command-line tool

Unicode solutions

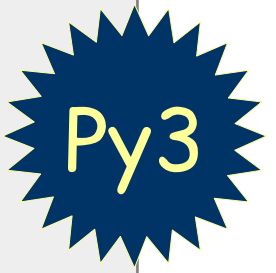


	0D0	0D1	0D2	0D3	0D4	0D5	0D6	0D7
0		ഐ 0D10	ഠ 0D20	ര 0D30	ീ 0D40		ൠ 0D60	൰ 0D70
1	ം 0D01		ഡ 0D21	റ 0D31	ു 0D41		ൡ 0D61	൱ 0D71
2	ഃ 0D02	അ 0D12	ഢ 0D22	ല 0D32	ൂ 0D42		ൢ 0D62	൲ 0D72
3	ഄ 0D03	ആ 0D13	ണ 0D23	ള 0D33	ൃ 0D43		ൣ 0D63	൳ 0D73
4		ഇ 0D14	ത 0D24	ഴ 0D34	ൄ 0D44			൴ 0D74
5	അ 0D05	ക 0D15	ഥ 0D25	വ 0D35				൵ 0D75
6	ആ 0D06	ഖ 0D16	ദ 0D26	ശ 0D36	ൊ 0D46		൦ 0D66	
7	ഇ 0D07	ഗ 0D17	ധ 0D27	ഷ 0D37	ോ 0D47	ൗ 0D57	൧ 0D67	
8	ഈ 0D08	ഘ 0D18	ന 0D28	സ 0D38	ൌ 0D48	ൈ	൨ 0D68	
9	ഉ 0D09	ങ 0D19	0D29	ഹ 0D39			൩ 0D69	൹ 0D79
A	ഊ 0D0A	ച 0D1A	പ 0D2A	0D3A	ൎ 0D4A		൪ 0D6A	ൺ 0D7A
B	ഋ 0D0B	ഛ 0D1B	ഫ 0D2B		൏ 0D4B		൫ 0D6B	ൻ 0D7B
C	ഌ 0D0C	ജ 0D1C	ബ 0D2C		൐ 0D4C		൬ 0D6C	ർ 0D7C
D		ഝ 0D1D	ഭ 0D2D	ഽ 0D3D	൑ 0D4D		൭ 0D6D	ൽ 0D7D
E	എ 0D0E	ഞ 0D1E	മ 0D2E	ാ 0D3E	0D4E		൮ 0D6E	ൾ 0D7E
F	ഏ 0D0F	ട 0D1F	യ 0D2F	ി 0D3F			൯ 0D6F	ൿ 0D7F

Combining characters

- Latin character accents and other diacritical marks can be written as separate characters

```
>>> s1 = 'café'
>>> s2 = 'cafe\u0301'
>>> import unicodedata
>>> unicodedata.name(s2[-1])
'COMBINING ACUTE ACCENT'
>>> print(s1, s2)
café café
>>> len(s1), len(s2)
(4, 5)
>>> list(s1), list(s2)
(['c', 'a', 'f', 'é'], ['c', 'a', 'f', 'e', '́'])
>>> s1 == s2
False
```



Normalization

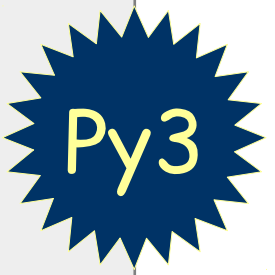
- Composing or decomposing all characters
  - Optional: replacing compatibility characters
- Normalization forms: NFC, NFKC, NFD, NFKD

```
>>> from unicodedata import normalize
>>> def nfc_equal(str1, str2):
...     return normalize('NFC', str1) == normalize('NFC', str2)
...
>>> s1 = 'café'
>>> s2 = 'cafe\u0301'
>>> s1 == s2
False
>>> nfc_equal(s1, s2)
True
```

Case folding

- Standard character substitutions

```
>>> def fold_equal(str1, str2):
...     return (normalize('NFC', str1).casefold() ==
...                 normalize('NFC', str2).casefold())
...
>>> s3 = 'Straße'
>>> s4 = 'strasse'
>>> s3 == s4
False
>>> nfc_equal(s3, s4)
False
>>> fold_equal(s3, s4)
True
>>> fold_equal(s1, s2)
True
>>> fold_equal('A', 'a')
True
```



2.7 gotcha:
unicode.casefold()
not implemented

Unicode sorting

- By default, text ordering uses the code point values of the characters
  - this is not what humans expect
    - in Portuguese, accents and diacritics are tiebreakers only

```
>>> fruits = ['caju', 'atemoia', 'cajá', 'açaí', 'acerola']
>>> sorted(fruits)
['acerola', 'atemoia', 'açaí', 'caju', 'cajá']
```

Py3

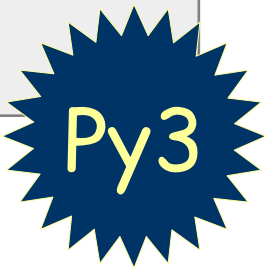
wrong: açaí should be first

wrong: caju should be last

Unicode sorting

- The standard library solution requires use of the **locale** module and a suitable locale available in the OS
  - only main program should set locale, and only at start-up
  - desired locale is not always available...

```
>>> import locale
>>> locale.setlocale(locale.LC_COLLATE, 'pt_BR.UTF-8')
'pt_BR.UTF-8'
>>> fruits = ['caju', 'atemoia', 'cajá', 'açaí', 'acerola']
>>> sorted_fruits = sorted(fruits, key=locale.strxfrm)
>>> sorted_fruits
['açaí', 'acerola', 'atemoia', 'cajá', 'caju']
```

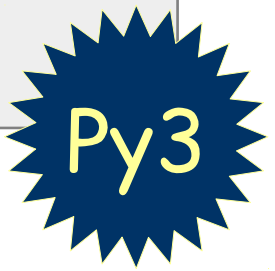


PythonPro

Unicode sorting

- James Tauber's PyUCA is a Python 3 implementation of UCA (Unicode Collation Algorithm)
  - locale independent!
  - designed to work with many languages
  - https://pypi.python.org/pypi/pyuca/

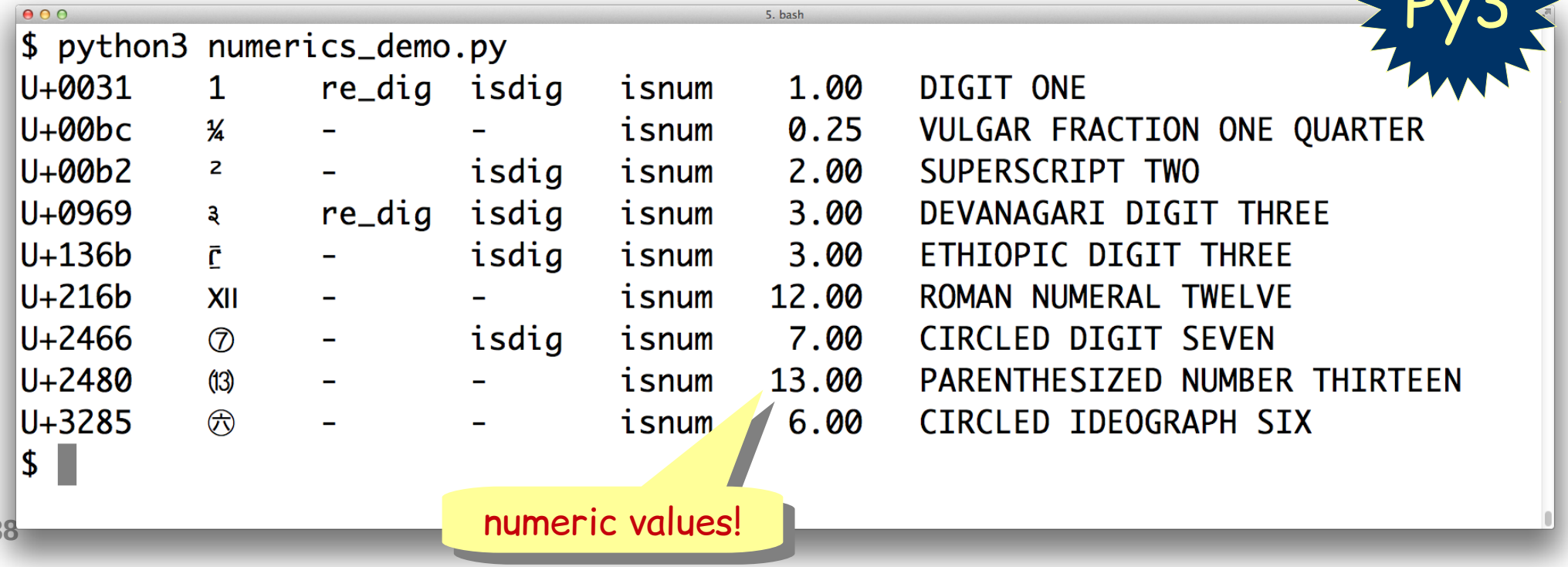
```
>>> import pyuca
>>> coll = pyuca.Collator()
>>> fruits = ['caju', 'atemoia', 'cajá', 'açaí', 'acerola']
>>> sorted_fruits = sorted(fruits, key=coll.sort_key)
>>> sorted_fruits
['açaí', 'acerola', 'atemoia', 'cajá', 'caju']
```



PythonPro

Unicode database

- Metadata about each Unicode character
  - name, numeric value, category etc.
  - standard library: **unicodedata** (Python 2 and Python 3)



Unicode database

5. bash

```
$ python3 numerics_demo.py
U+0031    1     re_dig  isdig   isnum     1.00   DIGIT ONE
U+00bc    ¼     -       -       isnum     0.25   VULGAR FRACTION ONE QUARTER
U+00b2    ²     -       isdig   isnum     2.00   SUPERSCRIPT TWO
U+0969    ३     re_dig  isdig   isnum     3.00   DEVANAGARI DIGIT THREE
U+136b    ፫     -       isdig   isnum     3.00   ETHIOPIC DIGIT THREE
U+216b    XII   -       -       isnum    12.00   ROMAN NUMERAL TWELVE
U+2466    ⑦     -       isdig   isnum     7.00   CIRCLED DIGIT SEVEN
U+2480    ⒀     -       -       isnum    13.00   PARENTHESIZED NUMBER THIRTEEN
U+3285    ㊅     -       -       isnum     6.00   CIRCLED IDEOGRAPH SIX
$
```

numerics_demo.py — Edited

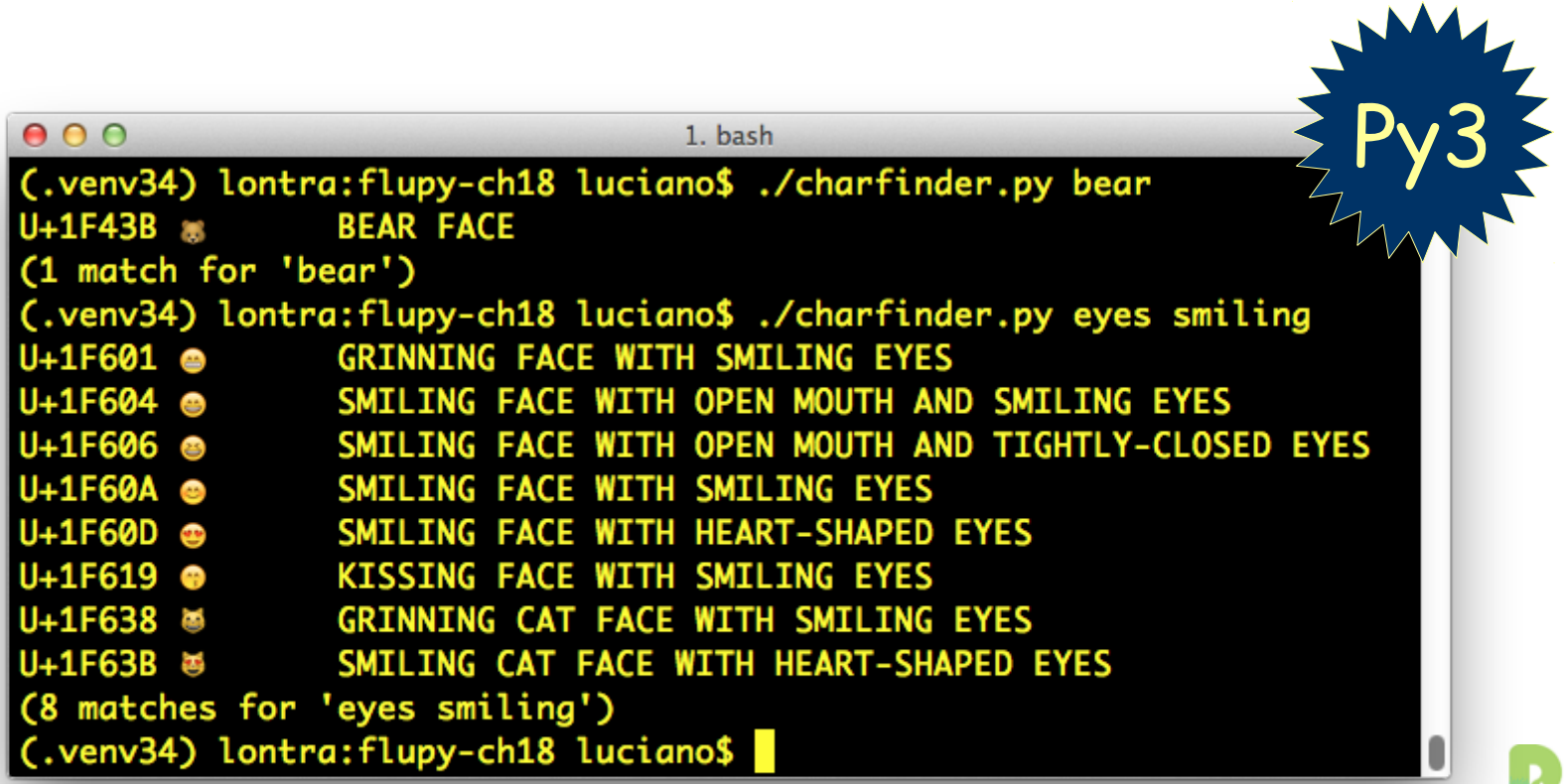
Py3

```
1  import unicodedata
2  import re
3
4  re_digit = re.compile(r'\d')
5
6  sample = '1\xbc\xb2\u0969\u136b\u216b\u2466\u2480\u3285'
7
8  for char in sample:
9      print('U+%04x' % ord(char),                          # <A>
10           char.center(6),                                # <B>
11           're_dig' if re_digit.match(char) else '-',     # <C>
12           'isdig' if char.isdigit() else '-',            # <D>
13           'isnum' if char.isnumeric() else '-',          # <E>
14           format(unicodedata.numeric(char), '5.2f'),     # <F>
15           unicodedata.name(char),                        # <G>
16           sep='\t')
17
```

Characters: 578 · Words: 57

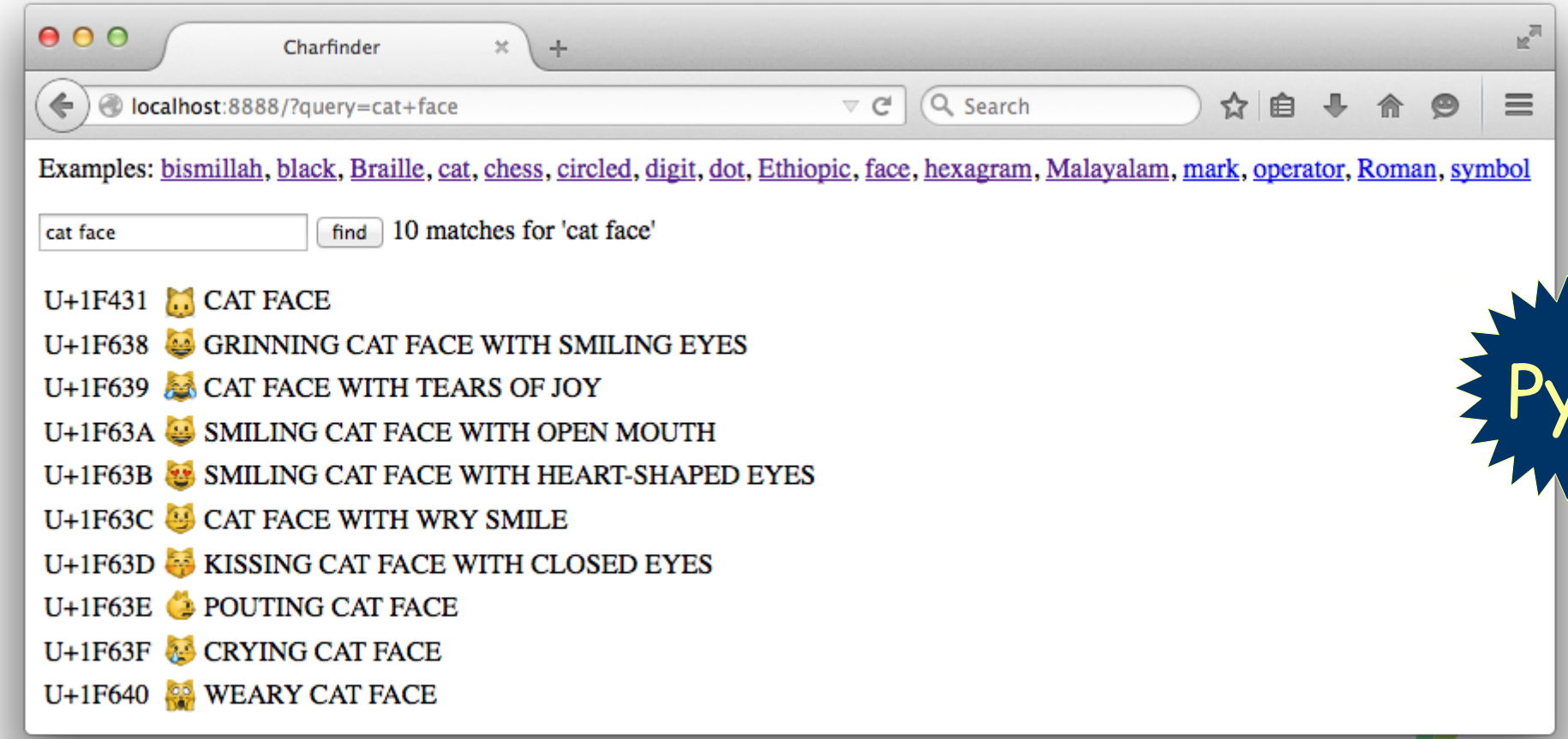
flupy-ch18/charfinder.py

- Command-line utility to search for characters by words in the official name
  - e.g. “cat face”, “black chess”...



flupy-ch18/http_charfinder.py

- HTTP and Telnet servers used to illustrate asyncio programming (*Fluent Python*, ch. 18)



Py3

¿Preguntas?

- More answers:
  - Python Unicode HOWTO
    - for Python 2
    - for Python 3
  - *Fluent Python*, chapter 4
  - Twitter: @ramalhoorg