CS2021
Week 4
Unicode, Files, and Directories
Strings and characters

The concept of “string” is simple enough: a string is a sequence of characters.

In 2015, the best definition of “character” we have is a Unicode character. Accordingly, the items you get out of a Python 3 str are Unicode characters.

The Unicode standard explicitly separates the identity of characters from specific byte representations: The identity of a character — a code point—is a number from 0 to 1,114,111 (base 10), shown in the Unicode standard as 4 to 6 hexadecimal digits with a “U+” prefix.

Unicode Code Points are characters

For example, the code point for:

- the Latin letter A is U+0041
- the Greek letter π is U+03C0
- the Euro sign is U+20AC
- the musical symbol G clef is U+1D11E.

About 10% of the valid code points have characters assigned to them in Unicode 6.3, the standard used in Python 3.4.

http://unicode.org/charts/

Armenian: Range: 0530–058F

……To……

Yi : Range: A000–A48F

Klingon is not yet in Unicode, but has used the “Private Use Areas”...

Encoding and Decoding

The actual bytes that represent a character depend on the encoding in use. An encoding is an algorithm that converts code points to byte sequences and vice versa. There are different encodings such as UTF-8 and UTF-16:

The code point for A (U+0041) is encoded:

-as the single byte in UTF-8: \x41

-as two bytes in UTF-16: \x41\x00

As another example, the Euro sign (U+20AC) is encoded:

- as three bytes in UTF-8: \xe2\x82\xac

- as two bytes but in UTF-16: \xac \x20.

Converting from code points to bytes is encoding; converting from bytes to code points is decoding.

Popular encodings

UTF-8 requires either 8, 16, 24 or 32 bits (one to four octets) to encode a Unicode character, UTF-16 requires either 16 or 32 bits to encode a character,

and UTF-32 always requires 32 bits to encode a character.

UTF-8 Generalizes ASCII (same 1 byte on first 128 chars) then uses 2 to 4 bytes for other codepoints.

UTF-16 always uses 2-4 bytes per codepoint and is more optimized for certain languages in 2-byte range

UTF-32 Fixed 4 bytes per letter (not supported in python)

Encoding examples

```
>>> s = 'café'
>>> type(s)
```

```
<class 'str'>
>>> len(s)   #=> 4
# The str 'café' has four Unicode
characters
# Encode str to bytes using UTF-8
encoding.
>>> b = s.encode('utf8')
>>> b
b'caf\xc3\xa9'
# b has five bytes (“é” is encoded as
two bytes in UTF-8).
>>> len(b)
5
>>> type(b)
<class 'bytes'>
>>>b.decode()
'café'
```

Unicode Encoding Errors

```
>>> f=open('/tmp/cafe.txt','w')
>>> f.write(s)
Traceback (most recent call last):
  File "<pyshell#28>", line 1, in
<module>
    f.write(s)
```

```
UnicodeEncodeError: 'ascii' codec can't
encode character '\xe9' in position 3:
ordinal not in range(128)

>>> f=open('/tmp/cafee.txt','w',
encoding='utf-8')
>>> f.write(s)
4


>>> f=open('/tmp/cafe.txt','r')
>>> ss=f.read()
Traceback (most recent call last):
  File "<pyshell#34>", line 1, in
<module>
    ss=f.read()
  File "/Library/Frameworks/
Python.framework/Versions/3.5/lib/
python3.5/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 3: ordinal
not in range(128)

>>> f=open('/tmp/
cafee.txt','r',encoding='utf-8')
```

```
>>> ss=f.read()
>>> ss
'café'
```

Opening files in binary mode

Not all files contain text. Some of them contain pictures of my dog.

```
>>> an_image = open('examples/tucker.jpg',
mode='rb')
>>> an_image.mode
'rb'
>>> an_image.name
'examples/tucker.jpg'
>>> an_image.encoding
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: '_io.BufferedReader' object has
no attribute 'encoding'
```

Opening a file in binary mode is simple but subtle. The only difference from opening it in text mode is that the mode parameter contains a 'b' character. Binary stream object has no encoding attribute.

You're reading (or writing) bytes, not strings, so there's no conversion for Python to do. What you get out of a binary file is exactly what you put into it, no conversion necessary.

Opening text in binary mode

```
>>> f=open('/tmp/
cafee.txt','br',encoding
='utf-8')
Traceback (most recent
call last):
  File "<pyshell#38>",
line 1, in <module>
    f=open('/tmp/
cafee.txt','br',encoding
='utf-8')
ValueError: binary mode
doesn't take an encoding
argument
>>> f=open('/tmp/
```

```
cafee.txt','br')
>>> data=f.read()
>>> data
b'caf\xc3\xa9'
>>> type(data)
<class 'bytes'>
>>> data.decode()
'café'
>>>
```

Filetools

Scanning filesystem

Using os.walk

```
import os
count=0

for (dirname,subdirs,files) in
os.walk('./PP4E'):
    for fi in files:
        p = os.path.join(dirname,fi)
```

```
        with
open(p,'r',encoding='utf-8') as f:
            data = f.read()
            if 'tkinter' in data: count
+=1
print("count", count)
```

The surrogateescape error handler

Still sometimes have errors:
UnicodeDecodeError: 'utf-8' codec can't decode byte 0x80 in position 3131: invalid start byte

If you know the encoding is ASCII-compatible and only want to examine or modify the ASCII parts, you can open the file with the surrogateescape error handler:

```
for fi in files:
        p = os.path.join(dirname,fi)
        f =
open(p,'r’,encoding="utf-8”,errors="surroga
```

```
teescape")
        data = f.read()
```

Problem: Find Largest Module

Suppose we want to find the largest module in our system

On my machine the standard modules are found in

/Library/Frameworks/Python.framework/Versions/3.5/lib/python3.5/

On windows you will find them in C:\Python31\Lib

Can you rind out where yours are?

```
# PP4E/System/Filetools/bigpy-tree.py
“”” Find the largest Python source file in an
entire directory tree.”””
import sys, os, pprint
trace = False
if sys.platform.startswith('win'):
    dirname = r'C:\Python31\Lib'
else:
     dirname = '/Library/Frameworks/
Python.framework/Versions/3.5/lib/python3.5/'
allsizes = []
for (thisDir, subsHere, filesHere) in
os.walk(dirname):
    if trace: print(thisDir)
    for filename in filesHere:
        if filename.endswith('.py'):
            if trace: print('...', filename)
            fullname = os.path.join(thisDir,
filename)
            fullsize = os.path.getsize(fullname)
            allsizes.append((fullsize, fullname))

allsizes.sort()
pprint.pprint(allsizes[:2])
pprint.pprint(allsizes[-2:])
```

```
# bigpy-path.py
# Now use all the directories on
sys.path, but skip
# visited directories
visited  = {}
```

```
allsizes = []
for srcdir in sys.path:
    for (thisDir, subsHere, filesHere)
in os.walk(srcdir):
         thisDir =
os.path.normpath(thisDir)
        fixcase =
os.path.normcase(thisDir)
        if fixcase in visited:
            continue
        else:
            visited[fixcase] = True
        for filename in filesHere:
            if
filename.endswith('.py'):
                pypath =
os.path.join(thisDir, filename)
                try:   “continued on
next slide”



try:
     pysize = os.path.getsize(pypath)
except os.error:
     print('skipping', pypath,
sys.exc_info()[0])
else:
```

```
        pylines = len(open(pypath,
'rb').readlines())
        allsizes.append((pysize, pylines,
pypath))
```

Homework #4

Modify your program from HW#3 to create a system utility that prints the largest and smallest files that contain an input keyword.

Add a third command line argument that indicates the number of smallest and largest files to print, so for example

$python myscript.py –k tucker –d / -n 5

Should print out the 5 smallest and largest files that contain

keyword tucker