String Operations

MARCH 19TH 2012

Python has lots of string operations…

- You can find a bunch of these in Section 5.6.1. "String Methods" of the Python documentation (v.2.7.2).

- These are in addition to the operations we studied that are common to lists *and* strings.
  - indexing, slicing
  - membership testing (`in` and `not in`) and concatenation (+).
  - `index`, `count`

String operations

- Here is a categorization (of some of these methods) that might help you navigate the long list of available string operations:
  - **Boolean methods**: isalpha, isalnum, isdigit, islower, isupper, isspace , startswith.
  - **Reformatting methods**: lower, upper, swapcase, capitalize, center, strip, lstrip, rstrip, ljust, rjust.
  - **Split methods**: split, lsplit, rsplit, splitlines.
  - **Join methods**: join.
  - **Replace methods**: replace

Examples: boolean methods

```
>>> "hello".isalpha()
True
>>> "hello".isalnum()
True
>>> "1234".isdigit()
True
>>> "39.78".isalnum()
False
>>> "hello?".islower()
True
>>> "Hello??".islower()
False
>>> "hello?".startswith("he")
True
>>> "hello?".startswith("He")
False
```

Examples: Reformatting methods

```
>>> "Hello, how are you?".lower()
'hello, how are you?'
>>> "Hello, how are you?".swapcase()
'hELLO, HOW ARE YOU?'
>>> "jack".capitalize()
'Jack'
>>> "   this string has spaces..   ".strip()
'this string has spaces..'
>>> "   this string has spaces..   ".lstrip()
'this string has spaces..   '
>>> "   this string has spaces..   ".rstrip()
'   this string has spaces..'
>>> "test".center(20)
'        test        '
>>> "hello??".rjust(20)
'            hello??'
```

Split and Join

```
>>> "hello, how are you?".split()
['hello,', 'how', 'are', 'you?']
>>> "Other, characters, can, be, used,,to,split?".split(",")
['Other', ' characters', ' can', ' be', ' used', '', 'to',
'split?']
>>> '''This string
... spans a
... few lines'''.splitlines()
['This string', 'spans a ', 'few lines']
>>> " ".join(["hello", "are","you","ok?"])
'hello are you ok?'
>>> "??".join(["hello", "are","you","ok?"])
'hello??are??you??ok?'
```

Replace

```
>>> "hello how are you?".replace(" ", "!")
'hello!how!are!you?‘
>>> "hello, how are you?".replace("h","")
'ello, ow are you?'
```

Problem: Creating a dictionary

Read a text file (e.g., a large novel such as “War and Peace”) that is guaranteed to consist of words that are correctly spelled. Extract “words” from this file and write these out in alphabetical order in a file called “dictionary.txt”.

A word is a contiguous sequence of letters, preceeded by a non-letter and followed by a non-letter.

Words in “dictionary.txt” should be unique and should be in lower case.

Extra credit: An attempt should be made to avoid proper nouns.

Version 1

```
#Programmer: Sriram Pemmaraju
#Date: March 13, 2012
#Version 1

# Open the input file "war.txt"
fin = open("smallWar.txt", "r")

wordList = []

# Loop that processes each line of the file
for line in fin:
   wordList = wordList + line.split()

#Close the input file
fin.close()

#Block of code that produces output
fout = open("dictionary1.txt", "w")
for word in wordList:
       fout.write(word+"\n")
fout.close()
```

Comments on Version 1

- I tried to run this on the full text of “War and Peace” but it was taking too long to complete. So I decided to run it on a smaller version of file consisting of just the first 1000 lines of the original.
- We will revisit the reason for this inefficiency later.
- The output file contains one word per line.
- But, the words contain non-letters and upper case letters.
- They are not in sorted order and surely contain duplicates.

Version 2

- I made three changes to Version 1 in order to deal with the issue of non-letters.
- First I created a list of all non-letter characters that might be in the text file.
- Notice the use of `ord`, `chr`, and `map` in this code.

```
# List of all non-letter characters
punctuationMarks = map(chr, range(0, ord("A")) + range(ord("Z")+1, ord("a")) +
range(ord("z")+1, 127))
```

Version 2

- I defined a function that takes a line (i.e., a string) and replaces every non-letter in this string by a blank.
- Notice the use of the replace method in this code.

```
# Replaces each non-letter character by a blank
def filterOutPunctuation(punctuationMarks, s):
    for mark in punctuationMarks:
        s = s.replace(mark, " ")
    return s
```

Version 2

- Finally, I process each line by first replacing non-letters by blanks and then splitting at blanks.

```
# Loop that processes each line of the file
for line in fin:
    newLine = filterOutPunctuation(punctuationMarks, line)
    wordList = wordList + newLine.split()
```

Comments on Version 2

- The list of words produced as output no longer contains non-letters.
- However, it does contain upper case letters, is not sorted, and contains duplicates!

Version 3

- I made three changes to Version 2 in order to deal with the upper case letter issue.
- First I defined a function that takes a list of words and turns this list into words in lower case.

```
def toLower(s):
   return s.lower()

def makeListLower(wordList):
    return map(toLower, wordList)
```

Version 3

- When each line is processed , the words are turned into lower case words.

```
# Loop that processes each line of the file
for line in fin:
    newLine = filterOutPunctuation(punctuationMarks, line)
    wordList = wordList + makeListLower(newLine.split())
```

Version 3

- The word list is sorted before being printed.

```
#Block of code that produces output
fout = open("dictionary3.txt", "w")
wordList.sort()
for word in wordList:
        fout.write(word+"\n")
fout.close()
```

Version 4

- Now I created a new version that dealt with the issue of duplicates.
- The only change I made is to the code that produces output.

```
fout = open("dictionary4.txt", "w")
wordList.sort()
previousWord = "" # keeps track of the word most recently output
for word in wordList:
    # only print out new words
    if word != previousWord:
        fout.write(word+"\n")
        previousWord = word
fout.close()
```

File I/O

- More often than not programs read from files, rather than from input typed at the keyboard.
- Often one program reads what another program outputs.
- More and more, programs are reading data produced by other hardware, e.g., sensors, telescopes, microarrays, etc.
- I these instances very little, if any, input is provided at the keyboard.

File objects

- Simplest Python statement for opening a file:

```
f = open("war.txt")
```

- Assuming that there is a file called "`war.txt`" in the same directory as your Python program, this statement *opens* the file for reading.

- Subsequently, the file can be accessed via the variable `f`.

- Since `f` is a variable, it has a type. Try `type(f)`.

File objects

- The variable f is often called a *file object*.
- If the file is missing from the directory, an error message is issued.

```
>>> g = open("hello.txt")
Traceback (most recent call last):
File "<string>", line 1, in <fragment>
IOError: [Errno 2] No such file or directory: 'hello.txt'
```

- One a file object is successfully connected to a file residing on your machine, we can use the file object to read from the file in a variety of ways.

Reading from a file

- `s = f.read()`
  Reads everything from the file into the string s

- `s = f.readline()`
  Reads the next line from the files into s

- ```
  for line in f:
      print line.split()
  ```
  Allows us to read and process the file line by line
  ```

Let us solve these problems on “War and Peace”

1. Build a dictionary of words extracted from the text that we might be able to use later, maybe in a spellchecker.

2. Compute the number of sentences in the text.

3. Compute the frequencies of letters in the text.

Two useful built-in Python functions that can help in solving Problem 3 are `ord` and `chr`.

Two useful functions

- `ord(ch)`

  if `ch` is a single character string, this function returns the ASCII code for `ch`

- `chr(i)`

  returns a string of one character whose ASCII code is the integer *i*

What is ASCII?

It stands for the *American Standard Code for Information Interchange*. It assigns a number in the range 0..255 to every character that can be entered at the keyboard.

More on ASCII

- The numbers 0..31 are reserved for unprintable characters, e.g., the tab character ("\t"), the end of line character ("\n"), etc.
- 32 is the ASCII value of the space character (" ")
- 33..47 is used for some punctuation characters
- 48..57 is used for digits "0" through "9"
- 65..90 is used for upper case letters
- 97..122 is used for lower case letters

ASCII Table

Dec	Hx	Oct	Char	Dec	Hx	Oct	Html	Chr	Dec	Hx	Oct	Html	Chr	Dec	Hx	Oct	Html	Chr
0	0	000	NUL (null)	32	20	040		Space	64	40	100	@	@	96	60	140	`	`
1	1	001	SOH (start of heading)	33	21	041	!	!	65	41	101	A	A	97	61	141	a	a
2	2	002	STX (start of text)	34	22	042	"	"	66	42	102	B	B	98	62	142	b	b
3	3	003	ETX (end of text)	35	23	043	#	#	67	43	103	C	C	99	63	143	c	c
4	4	004	EOT (end of transmission)	36	24	044	$	$	68	44	104	D	D	100	64	144	d	d
5	5	005	ENQ (enquiry)	37	25	045	%	%	69	45	105	E	E	101	65	145	e	e
6	6	006	ACK (acknowledge)	38	26	046	&	&	70	46	106	F	F	102	66	146	f	f
7	7	007	BEL (bell)	39	27	047	'	'	71	47	107	G	G	103	67	147	g	g
8	8	010	BS (backspace)	40	28	050	(	(	72	48	110	H	H	104	68	150	h	h
9	9	011	TAB (horizontal tab)	41	29	051	)	)	73	49	111	I	I	105	69	151	i	i
10	A	012	LF (NL line feed, new line)	42	2A	052	*	*	74	4A	112	J	J	106	6A	152	j	j
11	B	013	VT (vertical tab)	43	2B	053	+	+	75	4B	113	K	K	107	6B	153	k	k
12	C	014	FF (NP form feed, new page)	44	2C	054	,	,	76	4C	114	L	L	108	6C	154	l	l
13	D	015	CR (carriage return)	45	2D	055	-	-	77	4D	115	M	M	109	6D	155	m	m
14	E	016	SO (shift out)	46	2E	056	.	.	78	4E	116	N	N	110	6E	156	n	n
15	F	017	SI (shift in)	47	2F	057	/	/	79	4F	117	O	O	111	6F	157	o	o
16	10	020	DLE (data link escape)	48	30	060	0	0	80	50	120	P	P	112	70	160	p	p
17	11	021	DC1 (device control 1)	49	31	061	1	1	81	51	121	Q	Q	113	71	161	q	q
18	12	022	DC2 (device control 2)	50	32	062	2	2	82	52	122	R	R	114	72	162	r	r
19	13	023	DC3 (device control 3)	51	33	063	3	3	83	53	123	S	S	115	73	163	s	s
20	14	024	DC4 (device control 4)	52	34	064	4	4	84	54	124	T	T	116	74	164	t	t
21	15	025	NAK (negative acknowledge)	53	35	065	5	5	85	55	125	U	U	117	75	165	u	u
22	16	026	SYN (synchronous idle)	54	36	066	6	6	86	56	126	V	V	118	76	166	v	v
23	17	027	ETB (end of trans. block)	55	37	067	7	7	87	57	127	W	W	119	77	167	w	w
24	18	030	CAN (cancel)	56	38	070	8	8	88	58	130	X	X	120	78	170	x	x
25	19	031	EM (end of medium)	57	39	071	9	9	89	59	131	Y	Y	121	79	171	y	y
26	1A	032	SUB (substitute)	58	3A	072	:	:	90	5A	132	Z	Z	122	7A	172	z	z
27	1B	033	ESC (escape)	59	3B	073	;	;	91	5B	133	[	[	123	7B	173	{	{
28	1C	034	FS (file separator)	60	3C	074	<	<	92	5C	134	\	\	124	7C	174			|
29	1D	035	GS (group separator)	61	3D	075	=	=	93	5D	135	]	]	125	7D	175	}	}
30	1E	036	RS (record separator)	62	3E	076	>	>	94	5E	136	^	^	126	7E	176	~	~
31	1F	037	US (unit separator)	63	3F	077	?	?	95	5F	137	_	_	127	7F	177	&#127;	DEL

Some examples of `chr` and `ord` in action

```
>>> ord("a")
97
>>> chr(97)
'a'
>>> ord(" ")
32
>>> ord("0")
48
>>> chr(48)
'0'
>>> chr(49)
'1'
>>> ord("A")
65
>>> ord("B")
66
```

How are these functions useful?

- Because of the the fact that all the upper case letters occur consecutively in the ASCII table, the expression `ord(ch) – ord("A")` has value 0 for `ch= "A"`, value 1 for `ch = "B"`, has value 2 for `ch = "C"`, etc.

- Similarly, `ord(ch)-ord("a")` has value 0 for `ch = "a"` , has value 1 for `ch = "b"`, has value 2 for `ch = "c"`, etc.

A program to count letter frequencies

```
f = open("war.txt")
L = [0]*26
s = f.read()
for ch in s:
    if ch.isupper():
        L[ord(ch)-ord("A")] = L[ord(ch)-ord("A")] + 1
    elif ch.islower():
        L[ord(ch)-ord("a")] = L[ord(ch)-ord("a")] + 1
print L
```

Notice how ord(ch)-ord("A") and ord(ch)-ord("a") are used to index into the list L.

Another example

- The `ord` and `chr` functions can be used to perform Caeser's Cipher (Problem 3, HW 7).
- Try this: `chr(ord("a") + 4)`
- What does this expression evaluate to?