Week #6 - Python Practice

ZIP-fixing, Working With Text Data, and: Louie becomes Frodo!

Where we left off in our last Python Practice was fixing up the ZIP4 codes in the custZData data set. You'll recall that we observed that some ZIP4 code values were less than 4 digits long. You (probably) saved the various ZIP code Series we were working on, right? Let's get started. Crank up your Canopy, and import pandas, numpy, and DataFrame and Series from pandas. Then load the ZIP code Series we worked on. They were ZIP, ZIPlen, ZIP4, and ZIP4len. Did you save these ZIP Series as a DataFrame? By pickling? Well, we did. Here's how we load their DataFrame back into our session, assuming our pickle file is called pickledZIPs.p:

In [56]:
```
import os

import cPickle as pickle

import pandas as pd  # panda's nickname is pd

import numpy as np  # numpy as np

import sqlalchemy

from pandas import DataFrame, Series     # for convenience
from sqlalchemy import create_engine
from sqlalchemy import schema
```

In [57]:
```
ZIPFrame=pd.read_pickle('pickledZIPs.p')
```

In [58]:
```
ZIPFrame.shape
```

Out[58]: (30179, 4)

Looks like the DataFrame has the right number of rows and number of columns. The columns are:

In [59]:
```
ZIPFrame.columns =  ['ZIP','ZIPlen','ZIP4','ZIP4len']
ZIPFrame.head()
```

Out[59]:

	ZIP	ZIPlen	ZIP4	ZIP4len
0	60084	5	5016	4
1	60091	5	1750	4
2	60067	5	900	3
3	60068	5	3838	4
4	60090	5	3932	4

Remember that ZIP and ZIP4 are string, or character, variables. In the last Session Practice proposed to pad each ZIP4 value that is less than four digits long with zeros. We considered whether a four zero ZIP4, “0000”, should be a missing value. There are reasons why an address might not have a ZIP4 code. One reason is that not all addresses have been assigned one. Let's get ZIP4 out of the DataFrame to make them easier to work with:

In [60]:
```
ZIP4 = ZIPFrame.ZIP4            # Out with you, ZIP4's!
ZIP4len = ZIPFrame.ZIP4len
```

It turns out that pandas has many methods for examining and manipulating text strings. One example is the len() method we applied in the last Practice to calculate the number of characters in each customer's zip code. pandas doesn't have a method for padding strings with zeros, but it does have one for padding them with spaces. So,

In [61]:
```
ZIP4pad=ZIP4.str.pad(4)
type(ZIP4pad)
```

Out[61]:
```
pandas.core.series.Series
```

which should make all ZIP4's four characters long, including spaces:

In [62]:
```
ZIP4pad.str.len().value_counts()
```

Out[62]:
```
4    30179
Name: ZIP4, dtype: int64
```

Do all strings in ZIP4 now have a length of four? We “left-padded” strings with spaces if their length was less than four. Now let's change those spaces to zeros:

In [63]:
```
newZIP4=ZIP4pad.str.replace(' ','0')    # replace all spaces with zero
type(newZIP4)
```

Out[63]:
```
pandas.core.series.Series
```

In [64]:
```
newZIP4.head()
```

Out[64]:
```
0    5016
1    1750
2    0900
3    3838
4    3932
Name: ZIP4, dtype: object
```

Pretty easy, right? Let's take a look at some of those we've padded with zero, those with lengths of two, initially:

In [65]:
```
newZIP4[ZIP4len==2].head(5)
```

Out[65]:
```
51      0024
1633    0020
2567    0045
2803    0068
4219    0066
Name: ZIP4, dtype: object
```

There are some with all zeros, e.g.

In [66]:
```
newZIP4[newZIP4=='0000'].head(5)
```

Out[66]:
```
9     0000
11    0000
32    0000
41    0000
47    0000
Name: ZIP4, dtype: object
```

How many of them are there? Want to change these with '000' to missings?

In [67]:
```
newZIP4[newZIP4=='0000']=None
```

Did it work?

In [68]:
```
newZIP4[newZIP4.isnull()].head(5)
```

Out[68]:
```
9     None
11    None
32    None
41    None
47    None
Name: ZIP4, dtype: object
```

Since you have some previous experience with Python, you're probably aware of the variety of methods that can be applied to character strings in Python. There are many such methods. You can use them to change the case of letters in strings (lower to upper, upper to lower, to title, and so on), find characters, replace characters, split strings, format strings, perform logical tests on strings, and do many other things. The the time available for this Practice doesn't allow us to go into all aspects of strings in any detail, as we need to move on to what are called

“regular expressions” (see below) and some additional pandas capabilities. But you can look into the details of Python strings on you own by reviewing the official documentation that can be found at www.python.org or at other reputable properties. The textbooks by McKinney and Lubanovic provide good introductions to strings in Python. And, since everything in Python is an object, Python strings have attributes and methods. You can explore them by using the trick, e.g.

In [69]:
```
Boo='boo'
```

In [70]:
```
# Boo.<tab>
```

which will show you a number of string methods. String methods, like the regular expressions that follow below, are best understood by trying them out, by “fiddling” with them. Let's now consider a Python package called re. “re” is for “regular expression,” and a regular expression, AKA a “regex,” is a pattern that can be matched against data for searching, replacing, and filtering. Most all modern computer languages support the use of regexes. They are very often used in the processing of text data. How regexes are expressed varies a little across languages. And, regexes are a little like a language of their own. They can seem arcane and complex: "Some people, when confronted with a problem, think "I know, I'll use regular expressions." Now they have two problems." -Jamie Zawinski, 1997 But they can be very useful in many contexts. They are relatively easy to use in Python, and pandas has some regex methods. In what follows we explore regexes as they can be used in Python. Go ahead and import the re package into your Canopy session so we can try it out. Let's jump right in. First, we need some text to tinker with:

In [71]:
```
IMKool="I was too cool for school, \n but I signed up to do an MSPA an
nd now I'm no fool.So cool, \nso cool, so cool.\n"
```

In [72]:
```
IMKool
```

Out[72]:
```
"I was too cool for school, \n but I signed up to do an MSPA annd no
w I'm no fool.So cool, \nso cool, so cool.\n"
```

The \n's are newline characters. They'll work if you do print(IMKool). We can use a regex to see if there's a particular word, like “cool,” in IMKool:

In [73]:
```
import re
re.findall('cool', IMKool)      # 'cool' is the “regex”, the search pa
ttern, to find in IMKool
```

Out[73]:
```
['cool', 'cool', 'cool', 'cool']
```

A list of the “cool's” found is returned. The findall method of re finds all occurrences of the regex 'cool.' Try to find either “cool,” “school,” or “fool”:

In [74]:
```
re.findall('cool|school|fool',IMKool)
```

Out[74]:
```
['cool', 'school', 'fool', 'cool', 'cool', 'cool']
```

The “|” means “or.” Try finding words in a string of your own. The methods that re has for finding patterns in text do different things. The re methods match() and search() return “match objects” (which have methods: this is

Python), and finditer() returns an iterator that can be used for processing in loops. The re methods findall(), split(), and sub() return strings or a lists of strings. split() can divide up a string into list elements by using a regex. sub() does replacements by substituting a string for what matches a regex. subn() returns in tuples new strings after replacements along with a count of the number of replacements made in them. When using the methods match(), search(), findall(), and finditer() you can provide optional start and end arguments in numbers of characters ttelling them where in a string to begin and end their processing. Here's a simple example of using split(). Try:

```
In [75]: re.split(' ',IMKool)
```

```
Out[75]: ['I',
          'was',
          'too',
          'cool',
          'for',
          'school,',
          '\n',
          'but',
          'I',
          'signed',
          'up',
          'to',
          'do',
          'an',
          'MSPA',
          'annd',
          'now',
          "I'm",
          'no',
          'fool.So',
          'cool,',
          '\nso',
          'cool,',
          'so',
          'cool.\n']
```

That's a space between the single quotes. Doubles would also work, and:

```
In [76]: re.search('school',IMKool)
```

```
Out[76]: <_sre.SRE_Match at 0x10ff9b370>
```

returns something, an object, but

```
In [77]: re.match('school', IMKool)
```

give you nothing. The match() method returns nothing if it doesn't find its regex at the beginning of its source, its input string. Regexes are often compiled into bytecode if they are going to be used a lot, like by applying them to many sources:

In [78]:
```
schoolPat=re.compile('school')
```

In [79]:
```
re.findall(schoolPat,IMKool)
```

Out[79]:
```
['school']
```

Now try:

In [80]:
```
schoolPat=re.compile('school')
```

In [81]:
```
re.match(schoolPat, IMKool)
```

returns, nothing, but:

In [82]:
```
schoolPat=re.compile('.*school')                # note the .* before school.
```

In [83]:
```
obj=re.match(schoolPat,IMKool)
```

does return an object. Try the tab trick

In [84]:
```
# obj.<tab>
```

to see this object's attributes. One of them is the string that was matched:

In [85]:
```
obj.group()
```

Out[85]:
```
'I was too cool for school'
```

Regex's can include two kinds of characters. One kind is literal characters, like the “school” characters in the pattern '.*school”. The second kind is metacharacters. The characters “.” (dot) and “*” (asterisk) in this pattern are metacharacters. Metacharacters can be used to both generalize and constrain pattern matches. The dot (.) in this pattern means “any character,” and the * means any number of the proceeding, so the two together mean any number of any character before the string “school.” That's why what was returned by obj.group() was the entire string of characters up to the first occurrence of 'school.' The entire string matched the regex. Several metacharacters can be used in a regex. They include: • . (dot) matches any single character except \n • ? none or one of the preceding • * none, or one or more of the proceeding • + one or more of the preceding • {n,m} between n and m of the preceding • \w an alphanumeric character • \W a non-alphanumeric character • \d a single digit • \D a single non-digit • \s a single whitespace character • \S a single non-whitespace character • \b a boundary between a \w and a \W, or a \W and a \w • \B the opposite of \b: a non-boundary between \w and \W, or \W and \w • ^ matches the beginning of a line • $ the end of a line • \A the beginning of the input source • \Z the end of the input Let's try a couple of these. How about finding four letter words?

In [86]:
```
re.findall('\w{4}',IMKool)
```

Out[86]:
```
['cool', 'scho', 'sign', 'MSPA', 'annd', 'fool', 'cool', 'cool', 'cool']
```

In [87]:
```
re.findall('s.{,1}',IMKool)     # “s” followed by no more than one cha
racter
```

Out[87]:
```
['s ', 'sc', 'si', 'so', 'so']
```

Get the first whitespace character:

In [88]:
```
re.search('\s',IMKool).group()
```

Out[88]:
```
' '
```

What does this return? That group() method there at the end of the above re statement returns the first of results for what can be a number of grouped pattern “hits.” Grouped patterns are sub-patterns inside a main regex. They are represented by enclosing sub-patterns in parentheses, for example, compare:

In [89]:
```
socMatch=re.findall('\sso\sc',IMKool)
```

to

In [90]:
```
gsocMatch=re.findall('\s(so)\sc',IMKool)
```

See the difference? Remember that .findall() returns a list. In the second case, only elements with the pair 'so' that are followed by a space and the letter c are returned. Note that in either case, only “so's” were found. “So” wasn't. Why not? We could match both “so” and “So” by using what's called a character class or set. A set consists of items that are to be considered equivalent. Sets are demarked by square brackets, [and], that enclose them. For example.

In [91]:
```
soSocMatch=re.findall('[sS]o\sc', IMKool)
```

finds strings that are either 'so c' or 'So c'. There's a single space between the o and the c in these because of the \s, the single whitespace metacharacter. You may note that some metacharacters represent a predefined class. \w, for example, stands for [a-zA-z0-9_]. You can “negate” a set by using the caret character, the ^:

In [92]:
```
re.findall('[ab]','ababac')
```

Out[92]:
```
['a', 'b', 'a', 'b', 'a']
```

will return a list of a's and b's.

In [93]:
```
re.findall('[^ab]','ababac')
```

Out[93]:
```
['c']
```

will return things other than a or b. ^ inside set brackets negates. Outside it means the beginning of the line. So far we've seen the delimiters { }, ( ) and [ ]. To search for a pattern that includes these as literal characters, they need to be “escaped” in a regex, which is done by preceding them by a backslash, a \. Backslashes need to be escaped to make them literal, too, as do *, . (dot), ^, |, +, and ?. Twelve characters need to be escaped, altogether. Some of the metacharacters conflict with their meanings in Python, so it's common practice to use raw string notation when specifying a regex pattern, like: re.compile(r”\bSo”) where the small case “r” indicates that the patter is a raw string. Otherwise, in this example, Python would interpret the \b as a backspace. How

would you search for a backspace? A few other regex delimiters that can be useful are the following. “a” and “b” are characters: • a (?=b) means match a if it is followed by b • a (?1 b) match a if not followed by b • (?<= a) b match b if preceeded by a • (?

In [94]:
```
louieDF=pd.read_table('louielouie.txt',header=None)
```

Well, that was easy, wasn't it? The optional header=None argument indicates that the first record isn't a header record with variable or column names in it. You could use read_table or read_csv for this. Both have a plethora of methods and options that make them very flexible and effective. Check them out using the IPython documentation trick:

In [95]:
```
# pd.read_<tab>
```

So now we have Louie in a DataFrame, line by line. It has a single column, and the rows are the lines of the song. What's the current name of the column? How about changing it to 'Louie?'

In [96]:
```
louieDF.columns=['Louie']
```

Since there's only one column in louieDF, we can extract it to a Series to make working with it a little simpler:

In [97]:
```
louie=louieDF.Louie
```

Are you tired of the name Louie, yet? How about if we change it to another name throughout the song? A two syllable name would work given the melody. You can pick your own, but here we're going to use Frodo. We'll replace Louie in every line with Frodo using a vectorized string replacement method:

In [98]:
```
frodo = louie.str.replace('Louie','Frodo')
```

Are all the Louies now Frodos? You could look, of course, but if you had a lot of lines, looking would be inconvenient. You can, however, search for Louie:

In [99]:
```
louieMatches=frodo.str.match('Louie')
```

In [100]:
```
louieMatches.sum()
```

Out[100]: 0

frodo.str.match() on the right hand side of In [38] will return a boolean (True/False) vector. What do you think the value of louieMatches.sum() will be if there are no Louie's in frodo? If you happen to type 'frodo' on the command line, it may look like all the lines are right justified, i.e. that there are varying numbers of spaces on the left of each line. Are space characters really there? If so, what would you expect the lengths of the strings would be? If you happen to type 'frodo' on the command line, it may look like all the lines are right justified, i.e. that there are varying numbers of spaces on the left of each line. Are space characters really there? If so, what would you expect the lengths of the strings would be

In [101]:
```
frodo.str.len().value_counts().sort_index(ascending=False)
```

Out[101]:
```
38    1
36    1
33    2
32    1
31    2
29    2
28    1
27    2
25    2
22    1
21    1
20    7
18    5
11    6
10    1
Name: Louie, dtype: int64
```

The above calculates the number of characters in each record of frodo, including spaces and punctuation. It then counts up the number of different lengths (this is the value_counts() part), and then it sorts the results in descending order of length. The index for this Series is string lengths. You can use a regex with pandas' various text methods. For example, suppose we wanted to get string lengths based only on letters and numbers?

In [102]:
```
frodo.str.replace(r'[^a-zA-Z0-9]','').str.len() # '' is a pair of sing
le quotes
```

```
Out[102]: 0     14
          1      9
          2     14
          3     16
          4      9
          5     24
          6     22
          7     25
          8     22
          9     16
          10    13
          11    16
          12    14
          13    29
          14    21
          15    25
          16    22
          17    14
          18     9
          19    14
          20    16
          21     9
          22    26
          23    22
          24    23
          25    20
          26    26
          27    14
          28     9
          29    14
          30    16
          31     9
          32    17
          33    24
          34     6
          Name: Louie, dtype: int64
```

The regex is the r'[^a-zA-Z0-9]' which stands for all characters that are not a lower or upper case letter or a digit. Remember what the caret ^ inside set brackets does? How many times does Frodo's name now show up in the lyrics?

```
In [103]: frodo.str.findall(r'[fF]rodo').str.len().sum()
```

```
Out[103]: 16
```

The set or class [fF] will catch frodo with an upper case F or a lower case f. You could have used the .count() method instead of the above:

In [104]:
```
frodo.str.count(r'[fF]rodo').sum()
```

Out[104]: 16

Try the following to see what they return:

In [105]:
```
frodo.str.findall(r'[fF]rodo')
```

Out[105]:
```
0     [Frodo, Frodo]
1                 []
2                 []
3     [Frodo, Frodo]
4                 []
5                 []
6                 []
7                 []
8                 []
9     [Frodo, Frodo]
10                []
11    [Frodo, Frodo]
12                []
13                []
14                []
15                []
16                []
17    [Frodo, Frodo]
18                []
19                []
20    [Frodo, Frodo]
21                []
22                []
23                []
24                []
25                []
26                []
27    [Frodo, Frodo]
28                []
29                []
30    [Frodo, Frodo]
31                []
32                []
33                []
34                []
Name: Louie, dtype: object
```

In [106]:
```
frodo.str.count(r'[fF]rodo')
```

Out[106]:
```
0     2
1     0
2     0
3     2
4     0
5     0
6     0
7     0
8     0
9     2
10    0
11    2
12    0
13    0
14    0
15    0
16    0
17    2
18    0
19    0
20    2
21    0
22    0
23    0
24    0
25    0
26    0
27    2
28    0
29    0
30    2
31    0
32    0
33    0
34    0
Name: Louie, dtype: int64
```

Arrgh! A singing pirate wants to know that there are Aye Yi's in frodo:

In [107]:
```
frodo[frodo.str.contains(r'[Aa]ye-[Yy]i')]
```

Out[107]:
```
2     Aye-yi-yi-yi, I said
19    Aye-yi-yi-yi, I said
29    Aye-yi-yi-yi, I said
Name: Louie, dtype: object
```

Respect the pirate. How many did you find, Matey? Having fun, yet? pandas has many tools for examining and manipulating text data using its string (str) methods. Here we've looked at just a few of them using a Series of

text strings. Note that these methods can also be applied to DataFrame columns.

Deliverable #1: Print a list of sentences that have the words 'me' or 'Me'

In [108]:
```
frodo[frodo.str.contains(r'[Mm]e')]
```

Out[108]:
```
1                          Me gotta go
4                          Me gotta go
5        Fine little girl waits for me
8         Never know if I make it home
10                  Me gotta go, oh no
16       I smell the rose in her hair.
18                         Me gotta go
21                         Me gotta go
24    It won't be long, me see me love
28                         Me gotta go
31                         Me gotta go
Name: Louie, dtype: object
```

Deliverable #2: How many times [mM]e occured in frodo?

In [109]:
```
frodo.str.count(r'[Mm]e').sum()
```

Out[109]: 12

Deliverable #3: Replace every [mM]e occured in frodo by you?

In [110]:
```
frodo.str.replace(r'[Mm]e','you')
```

Out[110]:
```
0                          Frodo Frodo, oh no
1                                you gotta go
2                        Aye-yi-yi-yi, I said
3                        Frodo Frodo, oh baby
4                                you gotta go
5              Fine little girl waits for you
6                 Catch a ship across the sea
7             Sail that ship about, all alone
8               Never know if I make it hoyou
9                        Frodo Frodo, oh oh no
10                        you gotta go, oh no
11                       Frodo Frodo, oh baby
12                         I said we gotta go
13      Three nights and days I sail the sea
14                  Think of girl, constantly
15         On that ship, I dream she's there
16            I syoull the rose in her hair.
17                         Frodo Frodo, oh no
18                               you gotta go
19                       Aye-yi-yi-yi, I said
20                       Frodo Frodo, oh baby
21                               you gotta go
22    Okay, let's give it to 'em, right now!
23                See Jamaica, the moon above
24        It won't be long, you see you love
25                  Take her in my arms again
26         I tell her I'll never leave again
27                         Frodo Frodo, oh no
28                               you gotta go
29                       Aye-yi-yi-yi, I said
30                       Frodo Frodo, oh baby
31                               you gotta go
32                     I said we gotta go now
33           Let's take it on outta here now
34                                 Let's go!!
Name: Louie, dtype: object
```

In [ ]:
```

```