PYTHON LAB BOOK

Python For Programmers

UCSC Extension Online

Lab 8 Comprehensions

Topics

- Scope issues
- List comprehensions

v.4.0

lab07_1.py

```
#!/usr/bin/env python
"""lab07_1.py Provides a TranslateToKeypad function that,
when passed a string of alphanumeric characters, returns a
string of digits.
"""
def TranslateToKeypad(word):
    """Returns the word, translated to the telephone keypad equivalent:
    abc -> 2  ghi -> 4   mno -> 6   tuv -> 8
    def -> 3  jkl -> 5  pqrs -> 7  wxyz -> 9
    Other characters get passed on.
    """

    def KeyMap(ch):
        if ch in 'abcABC':
            return '2'
        if ch in 'defDEF':
            return '3'
        if ch in 'ghiGHI':
            return '4'
        if ch in 'jklJKL':
            return '5'
        if ch in 'mnoMNO':
            return '6'
        if ch in 'pqrsPQRS':
            return '7'
        if ch in 'tuvTUV':
            return '8'
        if ch in 'wxyzWXYZ':
            return '9'
        return ch

    translated_word = ''
    for ch in word:
        translated_word += KeyMap(ch)
    return translated_word



def main():
    """Tests the TranslateToKeypad function."""

    DATA = "peanut", "salt", "lemonade", "good time", ":10", "Zilch"
    for word in DATA:
        print "%10s -> %s" % (word, TranslateToKeypad(word))
```

```

if __name__ == "__main__":
    main()
"""
$ lab07_1.py
    peanut -> 732688
      salt -> 7258
  lemonade -> 53666233
 good time -> 4663 8463
       :10 -> :10
     zilch -> 94524
$"""
```

lab07_2.py

```
#!/usr/bin/env python
"""lab07_2.py -- Interactive identifier testing"""

import lab07_1

def main():
    while True:
        translate_this = raw_input("Word to translate: ")
        if translate_this == '':
            break
        print lab07_1.TranslateToKeypad(translate_this)

if __name__ == '__main__':
    main()

"""
$ lab07_2.py
Word to translate: diamond
3426663
Word to translate: Ruby
7829
Word to translate: zirconium
947266486
Word to translate:
$
"""
```

lab07_3.py

```
#!/usr/bin/env python
"""lab07_3.py
This program translates a line of text from English
to Pig Latin.  The rules for forming Pig Latin words
are as follows:
o  If the word begins with a vowel, add "way" to the
   end of the word.
o  If the word begins with a consonant, extract the
   consonants up to the first vowel, move those
   consonants to the end of the word, and add "ay".
"""

def Pigify(word):
    vowels = "aeiouyAEIOUY"

    if word[0] in vowels:
        return word + 'way'

    for i, char in enumerate(word):
        if char in vowels:
            break
    return word[i:] + word[:i] + 'ay'

def PrepWord(word):
    punctuations=",.:;!"
    if word[-1] in punctuations:
        word, punctuation = word[:-1], word[-1]
    else:
        punctuation = ''

    recase = 0
    if word.islower() or word.isupper():
        pass
    elif word[0].isupper():
        recase = 1
        word = word[0].lower() + word[1:]
    return recase, word, punctuation

def ToPig(word):
    """Returns a pig latin version of the word."""
    recase, bare_word, punctuation \
            = PrepWord(word)
    word =  Pigify(bare_word) + punctuation
    if recase:
        word = word[0].upper() + word[1:]
    return word

def PigLatinize(line):

    """Returns a string containing the pig latin version of the
    input line.
    """

    pigged_words = []
    for word in line.split():
        pigged_words += [ToPig(word)]
    return ' '.join(pigged_words)

def main():
    print PigLatinize(raw_input('Tell me something good: '))

if __name__ == '__main__':
    main()

"""
$ lab07_3.py
Tell me something good: Ice cream, hot fudge, nuts and a cherry.
Iceway eamcray, othay udgefay, utsnay andway away errychay.
$
"""
```

list_scopes.py
```
#!/usr/bin/env python
"""Scope issue with lists """

tea = ['earl grey', 'camomile', 'chai']

def AppendTea():
    tea.append('green') # <-- Good because we don't assign (bind)
                        #     the name, we use the function of
                        #     an existing name.
def AssignTea():
    global tea
    tea += ['blackberry']  # <- But for assignment, we need global
                           #    because it will try to bind the name
                           #    to the local namespace and fail.
def PlusAddTea():
    tea += ['peppermint']  #  <- No good

def main():
    AppendTea()
    print tea
    AssignTea()
    print tea
    PlusAddTea()

if __name__ == '__main__':
    main()
"""
$ list_scopes.py
['earl grey', 'camomile', 'chai', 'green']
['earl grey', 'camomile', 'chai', 'green', 'blackberry']
Traceback (most recent call last):
  File "./list_scopes.py", line 26, in <module>
    main()
  File "./list_scopes.py", line 23, in main
    PlusAddTea()
  File "./list_scopes.py", line 16, in PlusAddTea
    tea += ['peppermint']  #  <- No good
UnboundLocalError: local variable 'tea' referenced before assignment
$
"""
```

passing_sequences.py

```
#!/usr/bin/env python
"""Passing sequences."""

def ChangeNumber(number):
    number = 4
def ChangeString(string):
    string = "Boo"
def ChangeTuple(a_tuple):
    a_tuple = ('a', 'b')
def ChangeList(a_list):
    a_list = ['a', 'b']
def ChangeInList(a_list):
    a_list[1] = 'z'

def main():
    number = 3
    ChangeNumber(number)
    print "Numbers don't change:", number

    string = "Halloween"
    ChangeString(string)
    print "Strings don't change:", string

    a_tuple = 1, 2
    ChangeTuple(a_tuple)
    print "Tuples don't change:", a_tuple

    a_list = [1, 2, 3]
    ChangeList(a_list)
    print "Lists don't change:", a_list

    ChangeInList(a_list)
    print "Changing within lists does change:", a_list

if __name__ == '__main__':
    main()
"""
$ passing_sequences.py
Numbers don't change: 3
Strings don't change: Halloween
Tuples don't change: (1, 2)
Lists don't change: [1, 2, 3]
Changing within lists does change: [1, 'z', 3]
$ """
```

List Comprehensions and other fancy things

You already know that range returns a list of numbers:

```
>>> L = range(10)
>>> L
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

A list comprehension returns a fancier list:

```
>>> squares = [x**2 for x in range(10)]
>>> squares
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

The syntax of that was:

[**evaluates_to_one_element_using_x_or_not for x in sequence**]

You can add an `if` to filter some elements from the sequence:

```
>>> odds = [x for x in range(10) if x % 2]
>>> odds
[1, 3, 5, 7, 9]
```

Your list element can be a tuple:

```
>>> L_odd_tupled = [(x, x**2) for x in odds]
>>> L_odd_tupled
[(1, 1), (3, 9), (5, 25), (7, 49), (9, 81)]
```

You can add another `for` to the syntax to nest looping:

```
>>> V1 = range(1, 5)
>>> V1
[1, 2, 3, 4]
>>> V2 = range(10, 50, 10)
>>> V2
[10, 20, 30, 40]
>>> V_add = [x + y for x in V1 for y in V2] # x in V1 is outer
                                            # invariant loop
>>> V_add
[11, 21, 31, 41, 12, 22, 32, 42, 13, 23, 33, 43, 14, 24, 34, 44]
```

You can call a function:

```
>>> [str(x) for x in range(13)]
['0', '1', '2', '3', '4', '5', '6', '7', '8', '9', '10', '11', '12']
```

Useful constructs:

```
>>> [0 for x in range(5)]
[0, 0, 0, 0, 0]
>>> [[] for x in range(5)]
[[], [], [], [], []]
>>>
```

Like some other languages, Python also provides some sequence functions: `map()`, `filter()` and `reduce()`, and a `lambda` expression.

You give `reduce()` the name of a function that takes two arguments and a sequence. It gives you back the result of applying the function to the first 2 elements, then applying that result to the third, etc.

```
reduce(fn, seq) == fn( ... fn(fn(fn(seq[0], seq[1]), seq[2]), seq[3]), ...)
```

An example is easiest to see:

```
>>> def Add(x, y):
...     return x + y
...
>>> reduce(Add, range(1, 11))
55
```

So that was `1 + 2 + 3 + 4 + 5 + 6 + 7 + 8 + 9 + 10`

If you want to get all that in one line of code, you can use an *anonymous function* or *lambda expression.*

```
>>> reduce(lambda x,y: x + y, range(1,11))
55
```

But, for this example, simplest is:

```
>>> sum(range(1, 11))
55
```

For the next fancy thing, we have:

```
>> places = [1, 2, 3, 4]
>> chars = "abcde"
>> ords = (97, 98, 99, 100)
>> zip(places, chars, ords)
[(1, 'a', 97), (2, 'b', 98), (3, 'c', 99), (4, 'd', 100)]
>>
```

`zip()` takes any number of sequences and makes a list of tuples.

```
tuple[0] has (seq1[0], seq2[0], seq3[0], ...)
tuple[1] has (seq1[1], seq2[1], seq3[1], ...)
tuple[2] has (seq1[2], seq2[2], seq3[2], ...)
...
```

Notice that zip quits whenever any of the sequences is out of elements.

Maybe you'll use `zip()` to make the index into a sequence available in a for loop since `range(len(some_sequence))` gives you a list of the indices into the sequence:

```
>>> yums = 'chocolate', 'whipped cream', 'nuts'
>>> for (i, yum) in zip(range(len(yums)), yums):
...   print "yums[%d] = %s" % (i, yum)
...
yums[0] = chocolate
yums[1] = whipped cream
yums[2] = nuts
>>>
```

But there's an `enumerate` function since Python2.2.3 just for that purpose:

```
>>> for (i, yum) in enumerate(yums):
...   print "yums[%d] = %s" % (i, yum)
...
yums[0] = chocolate
yums[1] = whipped cream
yums[2] = nuts
>>>
```

Optional:

`filter()` and `map()` can always be replaced by a list comprehension and some style guides prefer that you use comprehensions, especially when it's easier to read.

So, these topics are optional:

`map()` takes a function that has one argument and applies it to each member of the sequence:

```
>>> def Sqr(x):
...     return x * x
...
>>> map(Sqr, range(10))
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

This also invites a `lambda` expression:

```
>>> map(lambda x:x**2, range(10))
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

Or a list comprehension:

```
>>> [x**2 for x in range(10)]
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

Any sequence is good, if it makes sense. Here we use a tuple:

```
>>> map(lambda x:x**2, (1,2,3))
[1, 4, 9]
```

Let's just multiply by 2 so we can use other sequences types:

```
>>> map(lambda x:x*2, ("hi","ho"))
['hihi', 'hoho']

>>> map(lambda x:x*2, "hi")
['hh', 'ii']
>>>
```

`filter()` needs a function that returns 1 or 0. It will return a sequence whose members passed a 1 back from the function:

```
>>> def IsEven(x):
...     return not x % 2
...
>>> filter(IsEven, range(10))
[0, 2, 4, 6, 8]
```

Again, a `lambda` expression is tempting:

```
>>> filter(lambda x:not x % 2, range(10))
[0, 2, 4, 6, 8]
```

But a list comprehension is more attractive:

```
>>> [x for x in range(10) if not x % 2]
[0, 2, 4, 6, 8]
```

This sort of programming is "Functional Programming"

Don't forget `range` alone for this example:

```
>>> range(0, 10, 2)
[0, 2, 4, 6, 8]
```

quiz.py
```
#!/usr/bin/env python
"""  Quiz 2 (Lab 08) answers
>>> x = 3
>>> y = x
>>> x = 8
>>> print y
3

>>> x = [1, 2, 3]
>>> y = x
>>> x[1] = 8
>>> print y
[1, 8, 3]

>>> x = (8, 88)
>>> y
[1, 8, 3]

>>> x = "123"
>>> y = x
>>> x = "abc"
>>> y
'123'
"""

print "\n2. Program \n"

for n in range(6):
    for m in range(6):
        print "%4d" % (n*m),
    print
"""
$ quiz.py

2. Program

   0    0    0    0    0    0
   0    1    2    3    4    5
   0    2    4    6    8   10
   0    3    6    9   12   15
   0    4    8   12   16   20
   0    5   10   15   20   25
$ """
```

Lab 08

1. Use a list comprehension to produce powers of 2: `[1, 2, 4, 8, 16, ...`

   How high must you go to get an error?

2. Use list comprehensions to make a function that returns a list of strings, each string emulating one card, and the whole list emulating a deck of cards. A test produces:

```
$ lab08_2.py
The deck contains:
2 of Clubs,  3 of Clubs,  4 of Clubs,  5 of Clubs,
6 of Clubs,  7 of Clubs,  8 of Clubs,  9 of Clubs,
10 of Clubs,  Jack of Clubs,  Queen of Clubs,  King of Clubs,
Ace of Clubs,  2 of Diamonds,  3 of Diamonds,  4 of Diamonds,
5 of Diamonds,  6 of Diamonds,  7 of Diamonds,  8 of Diamonds,
[some skipped]
Jack of Spades,  Queen of Spades,  King of Spades,  Ace of Spades,
Joker,  and Joker.
$
```

3. The goal of reducing the number of lines of code is never more important than producing readable code. However, for this exercise only, use list comprehensions to produce the quiz output again, this time with the fewest possible lines of code:

```
 0    0    0    0    0    0
 0    1    2    3    4    5
 0    2    4    6    8   10
 0    3    6    9   12   15
 0    4    8   12   16   20
 0    5   10   15   20   25
```

4. Write a function that expects a number as an argument and returns a string that represents the amount as money. For example:

```
         MakeMoneyString(3) -> $3.00
   MakeMoneyString(14.3123) -> $14.31
MakeMoneyString(1234567.89) -> $1,234,567.89
   MakeMoneyString(-88.888) -> -$88.89
```