homework-1

September 13, 2015

1 Introduction to Python - Homework #1

- Due Friday 9/18 at 1:45
- Submit on Courseworks - do not email

2 Academic Honesty

- The computer science department has strict polices. Check the department web page for details.
- Do not look at anybody else's source code. Do not show anybody your source, or leave your source where somebody could see it. You MUST write your own code.
- For this class, feel free to discuss issues with other people, but suggest waiting an hour or two after a discussion, before writing your code.
- Cases of non original source will be refered to the Judical Committee.

3 Tips

- Make SURE you are running Python 3.4, and not 2.7

```
In [91]: # the 3.4.3 is the version
         import sys
         sys.version

Out[91]: '3.4.3 |Anaconda 2.3.0 (x86_64)| (default, Mar  6 2015, 12:07:41) \n[GCC 4.2.1 (Apple Inc. buil
```

- isinstance

```
In [23]: # 'isinstance' is a little more concise then 'type' in some situations
         type(234) == int
         isinstance(234, int)

Out[23]: True
```

- 'list' will force a lazy eval

```
In [90]: e = enumerate(['a','b','c'])
         print(e)
         list(e)

<enumerate object at 0x107396c60>

Out[90]: [(0, 'a'), (1, 'b'), (2, 'c')]
```

- Lazy function types
- the 'type' of a lazy function's output is the name of the function

```
In [41]: [isinstance(range(10), range), isinstance(zip([0,1,2]),zip)]

Out[41]: [True, True]
```

- a common way to recurse thru a nested list is to split the list into the 0th element, and the rest of the list. 'rcount' shows an example

```
In [24]: # count the number of elements in a nested list
         def rcount(l):
             if l == []:
                 return(0)
             if isinstance(l, list):
                 return rcount(l[0]) + rcount(l[1:])
             else:
                 # not a list
                 return(1)

         rcount([3,4,[3,4,[4,2,3],3],3])

Out[24]: 9
```

- can set multiple vars from a list or tuple

```
In [5]: x, y, z = [1,2,3]
        x

Out[5]: 1

In [6]: z

Out[6]: 3
```

4 Problem 1a

- write a function(dp) that computes standard 'dot products'
- example: dp([1,2,3], [4,5,6]) =

$$1 * 4 + 2 * 5 + 3 * 6 = 32$$

- if one vector is longer than the other, the extra elements on the right are ignored

```
In [2]: # test vectors for Problem 1 a,b,c,d
        tv0 = [1,2,3]
        tv1 = [4,5,6,7,8,9]

In [4]: # the 7,8,9 elements are ignored
        dp(tv0, tv1)

Out[4]: 32
```

5 Problem 1b

- write 'shortlong', a function that takes two vectors, and returns in a list the short vector, the short vector length, the long vector, and the long vector length

```
In [8]: shortlong(tv0, tv1)

Out[8]: [[1, 2, 3], 3, [4, 5, 6, 7, 8, 9], 6]

In [9]: shortlong(tv1, tv0)

Out[9]: [[1, 2, 3], 3, [4, 5, 6, 7, 8, 9], 6]
```

6 Problem 1c

- write a more flexible version of dp, 'odp'
- odp takes an extra 'offset' arg, which moves the shorter vector to the right
- odp(tv0, tv1, 2) =

$$1 * 6 + 2 * 7 + 3 * 8 = 44$$

- use 'shortlong'

```
In [15]: [odp(tv0, tv1, 0), odp(tv0, tv1, 1), odp(tv0, tv1, 2)]

Out[15]: [32, 38, 44]
```

7 Problem 1d

- another version of dp, 'pdp'
- pdp takes a pad arg
- if one vector is shorter, it is padded on the right with the pad value
- pdp(tv0, tv1,1) = dp([1,2,3,1,1,1], [4,5,6,7,8,9])
- use 'shortlong'

```
In [21]: [pdp(tv0, tv1, 0), pdp(tv0, tv1, 1), pdp(tv0, tv1,2)]

Out[21]: [32, 56, 80]
```

8 Problem 2

- write a run length encoder
- convert a list into a list of [val, numberOfRepeats]

```
In [81]: rle([])

Out[81]: []

In [83]: rle([5])

Out[83]: [[5, 1]]

In [84]: rle([2,2])

Out[84]: [[2, 2]]

In [85]: rle([1,2])

Out[85]: [[1, 1], [2, 1]]

In [76]: rle([10,10,20,33,33,33,33,10,1,30,30,7])

Out[76]: [[10, 2], [20, 1], [33, 4], [10, 1], [1, 1], [30, 2], [7, 1]]
```

9 Problem 3

- write a function(partition), that divides a list into segments
- arg ‘l’ is the input list
- arg ‘n’ is the length of each segment. if there are not enough list elements to make a final segment of length n, they are discarded
- arg ‘overlap’ is how many list elements should overlap btw adjacent segments
- remember range is range(inclusive, exclusive), range[0,2] => [0,1]

```
In [43]: partition(range(10),2, 0)

Out[43]: [range(0, 2), range(2, 4), range(4, 6), range(6, 8), range(8, 10)]

In [48]: partition(list(range(10)), 2, 0)

Out[48]: [[0, 1], [2, 3], [4, 5], [6, 7], [8, 9]]

In [33]: partition(list(range(10)),2, 0)

Out[33]: [[0, 1], [2, 3], [4, 5], [6, 7], [8, 9]]

In [35]: partition(list(range(10)),2,1)

Out[35]: [[0, 1], [1, 2], [2, 3], [3, 4], [4, 5], [5, 6], [6, 7], [7, 8], [8, 9]]

In [36]: partition(list(range(10)),4)

Out[36]: [[0, 1, 2, 3], [4, 5, 6, 7]]

In [37]: partition(list(range(10)),4,3)

Out[37]: [[0, 1, 2, 3],
          [1, 2, 3, 4],
          [2, 3, 4, 5],
          [3, 4, 5, 6],
          [4, 5, 6, 7],
          [5, 6, 7, 8],
          [6, 7, 8, 9]]

In [38]: partition(range(10),4,3)

Out[38]: [range(0, 4),
          range(1, 5),
          range(2, 6),
          range(3, 7),
          range(4, 8),
          range(5, 9),
          range(6, 10)]
```

10 Problem 4a

- write ‘expandlazy’ - if given a ‘lazy’ range, zip, or enumerate, expand it into a list
- non lazy values are returned unchanged

```
In [63]: [expandlazy(234), expandlazy(range(3)), expandlazy('asdf'), expandlazy(enumerate(['a','b','c']

Out[63]: [234, [0, 1, 2], 'asdf', [(0, 'a'), (1, 'b'), (2, 'c')]]
```

11 Problem 4b

- write ‘expandlazylist’ - expand any lazy elements of a list

```
In [60]: ll = [1,2,3, range(4), 5, zip([1,2,3], [4,5]), 'asdf', enumerate(['a', 'b', 'c'])]
         ll
```

```
Out[60]: [1,
          2,
          3,
          range(0, 4),
          5,
          <zip at 0x10739b248>,
          'asdf',
          <enumerate at 0x107396948>]
```

```
In [62]: expandlazylist(ll)
```

```
Out[62]: [1,
          2,
          3,
          [0, 1, 2, 3],
          5,
          [(1, 4), (2, 5)],
          'asdf',
          [(0, 'a'), (1, 'b'), (2, 'c')]]
```

12 Problem 5

- ‘flatten’ turns a nested list into a linear one
- use recursion like ‘rcount’ above
- one way to think about it is it ‘erases’ all the brackets, except the two outermost ones

```
In [68]: flatten( [1,[2,3,4,[5,6,[7,8],9],11]])
```

```
Out[68]: [1, 2, 3, 4, 5, 6, 7, 8, 9, 11]
```

```
In [69]: flatten([1,2,3,[4,56],[44,55],7,8])
```

```
Out[69]: [1, 2, 3, 4, 56, 44, 55, 7, 8]
```