notebook_5_list_comprehensions

March 21, 2018

1 List Comprehensions

1.1 Objectives

1. Understand the list comprehension syntax
2. Demonstrate list processing with comprehensions
3. Use list comprehensions in probability simulations

1.2 List Comprehension

- Expression for constructing list
- Returns a new list
- Reads like math
  - Set builder notation

```
In [1]: mylist = [1,2,3,4,5]
        yourlist = [item ** 2 for item in mylist]
        yourlist
```

```
Out[1]: [1, 4, 9, 16, 25]
```

1.3 Building a Comprehension

1.4 Building a List Comprehension

1. Begin with an empty shell

2. Insert the input sequence

3. Give the elements a name

```
L = [   for    in       ]
L = [    for     in range(10)]
L = [    for num in range(10)]
```

```
In [2]: L = [num + 2 for num in range(10)]
        L
```

```
Out[2]: [2, 3, 4, 5, 6, 7, 8, 9, 10, 11]
```

1.5 Adding an optional filter

- The if portion is optional
- Syntax: `if boolean_cond`
  - After input sequence
- Only keeps value for which the condition is `True`

```
In [3]: L = [num + 2 for num in range(10) if num % 2 == 1]
        L

Out[3]: [3, 5, 7, 9, 11]
```

1.5.1 Comprehensions work on any input sequence

```
In [4]: # On string - gives list of characters
        [ch for ch in "Todd Iverson"]

Out[4]: ['T', 'o', 'd', 'd', ' ', 'I', 'v', 'e', 'r', 's', 'o', 'n']

In [5]: # On tuple - converts to list
        [item for item in (1,2,3)]

Out[5]: [1, 2, 3]

In [6]: # On a lazy sequence
        [tup for tup in enumerate(["a", "b", "c"])]

Out[6]: [(0, 'a'), (1, 'b'), (2, 'c')]
```

1.6 Writing Clean Code

1.6.1 Using helper functions

- **Clean Code Rule 1:** Use helper functions to hide complexity

```
In [7]: # Original comprehension
        [x**(1/3) for x in range(5) if x %2 == 1]

Out[7]: [1.0, 1.4422495703074083]

In [8]: # Helper functions
        is_odd = lambda x: x % 2 == 1
        cube_root = lambda x: x**(1/3)
        # Refactored comprehension
        [cube_root(x) for x in range(5) if is_odd(x)]

Out[8]: [1.0, 1.4422495703074083]
```

1.6.2 Exercise 1

Write a function `sum_of_squares` that computes the sum of the squares of the numbers in the input list. *Use a helper function in your solution*

Example `sum_of_squares([2, 3, 4]) === 4 + 9 + 16`

```
In [ ]:
```

1.6.3 Exercise 2

Write a function named `num_digits` that will return the number of digits in an integer.

Example `num_digits(1234) == 4`

Hint Using `str` and `len` might help!

```
In [ ]:
```

1.6.4 Exercise 3

Write a function called `sum_even` that sums up all the even numbers in a list. Include helper functions in your solution.

Example `sum_even([1,2,3,4]) == 6`

Hint You need a filter here.

```
In [ ]:
```

1.7 Unpacking multiple items

- Tuple unpacking assigns a name to each item
- Can be use in a comprehension on a sequence of tuples

```
In [9]: a, b, c = (1, 2, 3)
        b
```

```
Out[9]: 2
```

```
In [10]: l = ['a', 'b', 'c']
         m = [1, 2, 3]
         # Without unpacking
         [item for item in zip(l, m)]
```

```
Out[10]: [('a', 1), ('b', 2), ('c', 3)]
```

```
In [11]: # with unpacking
         [i + str(j) for i, j in zip(l, m)]
```

```
Out[11]: ['a1', 'b2', 'c3']
```

1.8 Matrices as list of lists

- We can represent a matrix with a list of lists

```
In [12]: m1 = [[1,2], [2,3], [4, 5]]
         m1
```

```
Out[12]: [[1, 2], [2, 3], [4, 5]]
```

```
In [13]: # You can use multiple lines (readability)
         m2 = [[1,2,3],
               [4,5,6]]
         m2
```

```
Out[13]: [[1, 2, 3], [4, 5, 6]]
```

1.9 Levels of Abstraction

- For nested data structures
- Each level describes the contents of that level
  - Ignore details of levels from above/below

1.10 Writing Clean Code

1.10.1 Working with Levels of Abstraction

- **Clean Code Rule 2:** Each function works on one, and only one, level of abstraction
  - Works with rule 1, using helper functions to hide complexity
- **Bottom-Up Programming**
  1. Start with the inner most level
  2. Work your way up the levels

1.11 Example

Write a function for matrix-scalar multiplication

```
In [14]: # Start with example and expressions
         # Function for atomic elements
         num = 1
         scalar = 5
         scalar_mult_item = num*scalar
         scalar_mult_item
```

```
Out[14]: 5
```

```
In [15]: scalar_mult_item = lambda scalar, num: num*scalar
         scalar_mult_item(1,5)
```

```
Out[15]: 5
```

Row function - a row is a list of numbers

```
In [16]: # Start with example and expression
         # Use the helper from the last level
         row = [1,2,3]
         scalar = 5
         new_row = [scalar_mult_item(scalar,num)
                    for num in row]
```

```
In [17]: # Convert expression to a lambda expression
         scalar_mult_row = lambda scalar, row: [scalar_mult_item(scalar,num)
                                                for num in row]
         scalar_mult_row(5, [1,2,3])
```

```
Out[17]: [5, 10, 15]
```

Matrix function - matrices are a list of rows

```
In [18]: # Start with an example and expressions
         # NOTE - we can use the last helper function
         mat = [[1,2,3],
                [4,5,6]]
         scalar = 5
         [scalar_mult_row(scalar, row)
          for row in mat]
```

```
Out[18]: [[5, 10, 15], [20, 25, 30]]
```

```
In [19]: # Convert expression into lambda
         scalar_mult = lambda scalar, mat: [scalar_mult_row(scalar, row)
                                            for row in mat]
         scalar_mult(5, mat)
```

```
Out[19]: [[5, 10, 15], [20, 25, 30]]
```

Package full solution in a `def` statement

- **Note** Hide the helper function inside the main function
- Use examples as test cases
  - Make sure they are correct!

```
In [20]: def scalar_mult(scalar, mat):
             """ Multiply a matrix of numbers mat by scalar"""
             scalar_mult_item = lambda scalar, num: num*scalar
             scalar_mult_row = lambda scalar, row: [scalar_mult_item(scalar,num)
                                                    for num in row]
             output = [scalar_mult_row(scalar, row)
                       for row in mat]
             return output
```

```
def test_scalar_mult():
    mat = [[1,2,3],
       [4,5,6]]
    s = 5
    assert scalar_mult(5, mat) == [[5, 10, 15], [20, 25, 30]]
test_scalar_mult()
```

1.12 Notes

- It is ok to skip the atomic function
  - For simple expressions
- Students resist this approach
  - Often the first hint they need

1.12.1 Exercise 4

Use a list comprehensions and lambda expression to create a sequence of functions that combine to average two matrices. A complete solution will provide functions for each level of abstraction. Package your results together in one function using a `def` statement and include a test function. **Be sure to obey Clean Code Rule 2!**

```
In [ ]:
```