Lecture 3.2 : List comprehensions

Introduction ¶

- Consider the following programming task: Write a Python function that accepts a list of integers as a parameter and returns a new list whose members are all the odd integers in input list. To solve such a programming task we might write code such as the following:

```
def extract_odds_v01(l):
    odds = []
    for n in l:
        if n % 2:
            odds.append(n)
    return odds
```

```
>>> import comps
>>> comps.extract_odds_v01([1, 2, 3, 4, 5, 6, 7, 8, 9, 10])
[1, 3, 5, 7, 9]
```

- It works. However it turns out that this pattern of building one list by processing (or transforming) the elements of another is so common that Python provides a short-cut for doing it. The short-cut is called a *list comprehension*.

List comprehensions

- A list comprehension is a short-cut to building one list from another. Its general form is: `[expression for-clause condition]`.
- The surrounding square brackets indicate we are building a list (i.e. they tell us this is a list comprehension). Let's refer to this list as `new_list`.
- The result of evaluating `expression` for each iteration of the `for-clause` is added to `new_list`. (It is typically an `expression` over elements of the list we are processing.)
- The `for-clause` visits each element of the list we are processing. Let's refer to this list as `old_list`.
- The `condition` allows us to select for inclusion in `new_list` expressions over only those elements of `old_list` that meet certain criteria. Unless the condition evaluates to true no new element is added to `new_list` and we move on to the next iteration of the `for-clause`.
- Rewriting our `extract_odds` function to use a list comprehension we get the code shown below. Note how compact it is compared to the original version.

```
def extract_odds_v02(l):
    return [n for n in l if n % 2]
```

```
>>> comps.extract_odds_v02([1, 2, 3, 4, 5, 6, 7, 8, 9, 10])
[1, 3, 5, 7, 9]
```

- The list comprehension `[n for n in l if n % 2]` can be read as:

  1. For each `n` in the list `l` (for-clause: `for n in l`)
  2. If `n` is odd (condition: `if n % 2`)
  3. Add `n` to the list being built (expression: `n`)

- Note how in this list comprehension `n` is used in all three of the `expression`, the `for-clause`, and the `condition`.

- Let's try another example. Write a Python function that accepts a string as a parameter and returns a new string whose characters are all those of the original string that are consonants. To solve this problem we could write code such as the following:

```
def extract_consonants_v01(s):
    consonants = []
    for c in s:
        if c.lower() not in 'aeiou':
            consonants.append(c)
    return ''.join(consonants)
```

```
>>> comps.extract_consonants_v01('Are vowels required to understand sentences?'
'r vwls rqrd t ndrstnd sntncs?'
```

- Rewriting the code to use a list comprehension we get:

```
def extract_consonants_v02(s):
    return ''.join([c for c in s if c.lower() not in 'aeiou'])
```

```
>>> comps.extract_consonants_v02('Are vowels required to understand sentences?'
'r vwls rqrd t ndrstnd sntncs?'
```

- The list comprehension `[c for c in s if c.lower() not in 'aeiou']` can be read as:

  1. For each `c` in the string `s` (for-clause: `for c in s`)
  2. If `c` is a vowel (condition: `if c.lower() not in 'aeiou'`)
  3. Add `c` to the list being built (expression: `c`)

- Let's try another example. Write a Python function that accepts a list of integers as a parameter and returns a new list whose members are the square of all the even integers in the input list. To solve this problem we could write code such as the following:

```
def even_squares_v01(l):
    squares = []
    for n in l:
        if not n % 2:
            squares.append(n**2)
    return squares
```

```
>>> comps.even_squares_v01([1, 2, 3, 4, 5, 6, 7, 8, 9, 10])
[4, 16, 36, 64, 100]
```

- Rewriting the code to use a list comprehension we get:

```
def even_squares_v02(l):
    return [n**2 for n in l if not n % 2]
```

```
>>> comps.even_squares_v02([1, 2, 3, 4, 5, 6, 7, 8, 9, 10])
[4, 16, 36, 64, 100]
```

- Let’s try a final example. Write a Python function that accepts a list of integers as a parameter and returns a new list whose members are the square of all the even integers in the input list and the cube of all the odd integers in the input list. To solve this problem we could write code such as the following:

```
def even_squares_odd_cubes_v01(l):
    squares_n_cubes = []
    for n in l:
        if not n % 2:
            squares_n_cubes.append(n**2)
        else:
            squares_n_cubes.append(n**3)

    return squares_n_cubes
```

```
>>> comps.even_squares_odd_cubes_v01([1, 2, 3, 4, 5, 6, 7, 8, 9, 10])
[1, 4, 27, 16, 125, 36, 343, 64, 729, 100]
```

- Rewriting the code to use a list comprehension we get:

```
def even_squares_odd_cubes_v02(l):
    return [n**2 if not n % 2 else n**3 for n in l]
```

```
>>> comps.even_squares_odd_cubes_v02([1, 2, 3, 4, 5, 6, 7, 8, 9, 10])
[1, 4, 27, 16, 125, 36, 343, 64, 729, 100]
```

- If you research Python solutions to programming problems on-line you will find they often make use of comprehensions so it is important you understand how they work. Using comprehensions where appropriate is considered the *Pythonic* way of problem-solving. When you move on to study other programming languages however you will find they do not support such comprehension constructs so you will revert to our original pattern to building one list from another.

More comprehensions

- Comprehensions do not apply to lists alone. They can be used as short-cuts to building a new *collection* from another *collection*. Thus it makes sense to not only talk about list comprehensions but also *set* and *dictionary comprehensions*. We may look at these later in the course.