PyLearn_Comprehension

May 4, 2020

#

Comprehension

0.1 GeeksforGeeks: Comprehensions in Python

https://www.geeksforgeeks.org/comprehensions-in-python/

4 types of comprehensions: - List Comprehensions - Dictionary Comprehensions - Set Comprehensions - Generator Comprehensions

0.1.1 List Comprehensions

basic structure of a list comprehension:

`output_list = [output_exp for var in input_list if (var satisfies this condition)]`

- may or may not contain an `if` condition.
- can contain multiple `for` (nested list comprehensions).

Example #1

[5]:
```
# Constructing output list WITHOUT
# Using List comprehensions

input_list = [1, 2, 3, 4, 4, 5, 6, 7, 7]

output_list = []
# Using loop for constructing output list
for var in input_list:
    if var % 2 == 0:
        output_list.append(var)

print("Output List using for loop:", output_list)
```

```
Output List using for loop: [2, 4, 4, 6]
```

[6]:
```
# Using List comprehensions
# for constructing output list
```

```
input_list = [1, 2, 3, 4, 4, 5, 6, 7, 7]

list_using_comp = [var for var in input_list if var % 2 == 0]

print("Output List using list comprehensions:", list_using_comp)
```

```
Output List using list comprehensions: [2, 4, 4, 6]
```

Example #2

[7]:
```
# Constructing output list using for loop
output_list = []
for var in range(1, 10):
    output_list.append(var ** 2)

print("Output List using for loop:", output_list)
```

```
Output List using for loop: [1, 4, 9, 16, 25, 36, 49, 64, 81]
```

[35]:
```
# Constructing output list
# using list comprehension
list_using_comp = [var**2 for var in range(1, 10)]

print("Output List using list comprehension:", list_using_comp)
```

```
Output List using list comprehension: [1, 4, 9, 16, 25, 36, 49, 64, 81]
```

0.1.2 Dictionary Comprehensions

The basic structure of a dictionary comprehension looks like below.

```
output_dict = {key:value for (key, value) in iterable if (key, value satisfy this condition)}
```

Example 1

[14]:
```
input_list = [1, 2, 3, 4, 5, 6, 7]

output_dict = {}

# Using loop for constructing output dictionary
for var in input_list:
    if var % 2 != 0:
        output_dict[var] = var**3

print("Output Dictionary using for loop:", output_dict)
```

```
Output Dictionary using for loop: {1: 1, 3: 27, 5: 125, 7: 343}
```

[17]:
```
# Using Dictionary comprehensions
# for constructing output dictionary

input_list = [1,2,3,4,5,6,7]

dict_using_comp = {var:var ** 3 for var in input_list if var % 2 != 0}

print("Output Dictionary using dictionary comprehensions:", dict_using_comp)
```

```
Output Dictionary using dictionary comprehensions: {1: 1, 3: 27, 5: 125, 7: 343}
```

Example 2

[36]:
```
state = ['Gujarat', 'Maharashtra', 'Rajasthan']
capital = ['Gandhinagar', 'Mumbai', 'Jaipur']

output_dict = {}

# Using loop for constructing output dictionary
for (key, value) in zip(state, capital):
    output_dict[key] = value

print("Output Dictionary using for loop:", output_dict)
```

```
Output Dictionary using for loop: {'Gujarat': 'Gandhinagar', 'Maharashtra':
'Mumbai', 'Rajasthan': 'Jaipur'}
```

[37]:
```
# Using Dictionary comprehensions
# for constructing output dictionary

state = ['Gujarat', 'Maharashtra', 'Rajasthan']
capital = ['Gandhinagar', 'Mumbai', 'Jaipur']

dict_using_comp = {key:value for (key, value) in zip(state, capital)}

print("Output Dictionary using dictionary comprehensions:", dict_using_comp)
```

```
Output Dictionary using dictionary comprehensions: {'Gujarat': 'Gandhinagar',
'Maharashtra': 'Mumbai', 'Rajasthan': 'Jaipur'}
```

0.1.3 Set Comprehension

Set comprehensions are pretty similar to list comprehensions. The only difference between them is that set comprehensions use curly brackets { }.

Example 1

[21]:
```
input_list = [1, 2, 3, 4, 4, 5, 6, 6, 6, 7, 7]

output_set = set()

# Using loop for constructing output set
for var in input_list:
    if var % 2 == 0:
        output_set.add(var)

print("Output Set using for loop:", output_set)
```

```
Output Set using for loop: {2, 4, 6}
```

[22]:
```
# Using Set comprehensions
# for constructing output set

input_list = [1, 2, 3, 4, 4, 5, 6, 6, 6, 7, 7]

set_using_comp = {var for var in input_list if var % 2 == 0}

print("Output Set using set comprehensions:", set_using_comp)
```

```
Output Set using set comprehensions: {2, 4, 6}
```

0.1.4 Generator Comprehensions

Generator Comprehensions are very similar to list comprehensions. One difference between them is that generator comprehensions use circular brackets whereas list comprehensions use square brackets. The major difference between them is that generators don't allocate memory for the whole list. Instead, they generate each value one by one which is why they are memory efficient.

[24]:
```
input_list = [1, 2, 3, 4, 4, 5, 6, 7, 7]

output_gen = (var for var in input_list if var % 2 == 0)

print("Output values using generator comprehensions:", end = ' ')

for var in output_gen: print(var, end = ' ')
```

```
Output values using generator comprehensions: 2 4 4 6
```

0.2 Nested List Comprehensions in Python

https://www.geeksforgeeks.org/nested-list-comprehensions-in-python/?ref=rp

0.2.1 Example 1

We want to create a matrix which looks like below:

```
matrix = [[0, 1, 2, 3, 4],
          [0, 1, 2, 3, 4],
          [0, 1, 2, 3, 4],
          [0, 1, 2, 3, 4],
          [0, 1, 2, 3, 4]]
```

[42]:
```
# Using nested for loop
matrix = []

for i in range(5):

    # Append an empty sublist inside the list
    matrix.append([])

    for j in range(5):
        matrix[i].append(j)

print(matrix)
```

```
[[0, 1, 2, 3, 4], [0, 1, 2, 3, 4], [0, 1, 2, 3, 4], [0, 1, 2, 3, 4], [0, 1, 2,
3, 4]]
```

[43]:
```
# Using Nested list comprehension
matrix = [[j for j in range(5)] for i in range(5)]

print(matrix)
```

```
[[0, 1, 2, 3, 4], [0, 1, 2, 3, 4], [0, 1, 2, 3, 4], [0, 1, 2, 3, 4], [0, 1, 2,
3, 4]]
```

- `[<expression> for i in range(5)]` –> which means that execute this `<expression>` and append its output to the list until variable i iterates from 0 to 4.

- For example:- `[i for i in range(5)]` –> In this case, the output of the expression is simply the variable i itself and hence we append its output to the list while i iterates from 0 to 4. Thus the output would be –> `[0, 1, 2, 3, 4]`
- But in our case, the `<expression>` itself is a list comprehension. Hence we need to first solve the expression and then append its output to the list.
- `<expression>` is `[j for j in range(5)]` –> The output of this expression is same as the example discussed above. Hence `<expression>` is `[0, 1, 2, 3, 4]`.
- Now we just simply append this output until variable `i` iterates from 0 to 4 which would be total 5 iterations. Hence the final output would just be a list of the output of the above `<expression>` repeated 5 times. Output: `[[0, 1, 2, 3, 4], [0, 1, 2, 3, 4], [0, 1, 2, 3, 4], [0, 1, 2, 3, 4], [0, 1, 2, 3, 4]]`

0.2.2 Example 2

Suppose we want to flatten a given 2-D list:

`matrix = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]`

Expected Output: `flatten_matrix = [1, 2, 3, 4, 5, 6, 7, 8, 9]`

[46]:
```
# Using nested for loop
# 2-D List
matrix = [[1, 2, 3], [4, 5], [6, 7, 8, 9]]

flatten_matrix = []

for sublist in matrix:
    for val in sublist:
        flatten_matrix.append(val)

print(flatten_matrix)
```

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

[50]:
```
# Using Nested list comprehension
# 2-D List
matrix = [[1, 2, 3], [4, 5], [6, 7, 8, 9]]

# Nested List Comprehension to flatten a given 2-D matrix
flatten_matrix = [val for sublist in matrix for val in sublist]

print(flatten_matrix)
```

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

- In this case, we need to loop over each element in the given 2-D list and append it to another list.

- For better understanding, we can divide the list comprehension into three parts:

```
flatten_matrix = [val
for sublist in matrix
for val in sublist]
```

- The first line suggests what we want to append to the list.

- The second line is the **outer** loop
- The third line is the **inner** loop
- `for sublist in matrix` returns the sublists inside the matrix one by one which would be: `[1, 2, 3], [4, 5], [6, 7, 8, 9]`
- `for val in sublist` returns all the values inside the sublist.

- Hence if sublist = `[1, 2, 3]`, `for val in sublist` gives `1, 2, 3` as output one by one.
- For every such `val`, we get the output as `val` and we append it to the list.

0.2.3 Example 3

Suppose we want to flatten a given 2-D list and only include those strings whose lengths are less than 6:

```
planets = [[‘Mercury’, ‘Venus’,   ‘Earth’],
           [‘Mars’,    ‘Jupiter’, ‘Saturn’],
           [‘Uranus’,  ‘Neptune’, ‘Pluto’]]
```

Expected Output: flatten_planets is `[‘Venus’, ‘Earth’, ‘Mars’, ‘Pluto’]`

[48]:
```
# Using an if condition inside a nested for loop
# 2-D List of planets
planets = [['Mercury', 'Venus', 'Earth'], ['Mars', 'Jupiter', 'Saturn'],␣
 ↪['Uranus', 'Neptune', 'Pluto']]

flatten_planets = []

for sublist in planets:
    for planet in sublist:

        if len(planet) < 6:
            flatten_planets.append(planet)

print(flatten_planets)
```

```
['Venus', 'Earth', 'Mars', 'Pluto']
```

[49]:
```
# Using Nested list comprehension
# 2-D List of planets
planets = [['Mercury', 'Venus', 'Earth'], ['Mars', 'Jupiter', 'Saturn'],␣
 ↪['Uranus', 'Neptune', 'Pluto']]

# Nested List comprehension with an if condition
flatten_planets = [planet for sublist in planets for planet in sublist if␣
 ↪len(planet) < 6]

print(flatten_planets)
```

```
['Venus', 'Earth', 'Mars', 'Pluto']
```

- This example is quite similar to the previous example but in this example, we just need an extra `if` condition to check if the length of a particular planet is less than 6 or not.
- This can be divided into 4 parts as follows:

```
flatten_planets = [planet
for sublist in planets
for planet in sublist
if len(planet) < 6]
```

0.3 Trey Hunner: Python List Comprehensions: Explained Visually

https://treyhunner.com/2015/12/python-list-comprehensions-now-in-color/

[55]:
```
numbers = [1, 5, 4, 2, 7, 6]

doubled_odds = list( map(lambda n: n * 2, filter(lambda n: n % 2 == 1,␣
↪numbers)) )
print(doubled_odds)

doubled_odds = [n * 2 for n in numbers if n % 2 == 1]
print(doubled_odds)
```

```
[2, 10, 14]
[2, 10, 14]
```

0.3.1 From loops to comprehensions

- Every list comprehension can be rewritten as a `for` loop but not every `for` loop can be rewritten as a list comprehension.
- The key to understanding when to use list comprehensions is to practice identifying problems that smell like list comprehensions.
- If you can rewrite your code to look just like this `for` loop, you can also rewrite it as a list comprehension:

```
new_things = []
for ITEM in old_things:
    if condition_based_on(ITEM):
        new_things.append("something with " + ITEM)
```

```
You can rewrite the above `for` loop as a list comprehension like this:

    new_things = ["something with " + ITEM for ITEM in old_things if condition_based_on(ITEM)]
```

0.3.2 Nested Loops

[62]:
```
matrix = [[1, 2, 3], [4, 5], [6, 7, 8, 9]]

flattened = []
for row in matrix:
    for n in row:
        flattened.append(n)
```

```
print(flattened)
```

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

[69]:
```
matrix = [[1, 2, 3], [4, 5], [6, 7, 8, 9]]

# flattened = [n for n in row for row in matrix]  # Will produce ERROR !!!!

flattened = [n for row in matrix for n in row]
print(flattened)
```

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

When working with nested loops in list comprehensions remember that the `for` clauses **remain in the same order** as in our original `for` loops.

0.3.3 Other Comprehensions (set comprehensions and dictionary comprehensions)

Set comprehensions

[75]:
```
words = ['Mercury', 'Venus', 'Earth', 'Mars', 'Jupiter', 'Saturn', 'Uranus',␣
↪'Neptune', 'Pluto']

first_letters = set()
for w in words:
    first_letters.add(w[0])

print(first_letters)
```

```
{'S', 'V', 'M', 'P', 'U', 'E', 'J', 'N'}
```

[76]:
```
words = ['Mercury', 'Venus', 'Earth', 'Mars', 'Jupiter', 'Saturn', 'Uranus',␣
↪'Neptune', 'Pluto']

first_letters = {w[0] for w in words}

print(first_letters)
```

```
{'S', 'V', 'M', 'P', 'U', 'E', 'J', 'N'}
```

Dictionary comprehensions

[80]:
```
original = {'one': 1, 'two': 2, 'three': 3}

flipped = {}
for key, value in original.items():
    flipped[value] = key
```

```
print(flipped)
```

```
{1: 'one', 2: 'two', 3: 'three'}
```

[81]:
```
original = {'one': 1, 'two': 2, 'three': 3}

flipped = {value: key for key, value in original.items()}
print(flipped)
```

```
{1: 'one', 2: 'two', 3: 'three'}
```

Overusing list comprehensions and generator expressions in Python
https://treyhunner.com/2019/03/abusing-and-overusing-list-comprehensions-in-python/

Trey Hunner's youtube: Comprehensible Comprehensions
https://www.youtube.com/watch?v=5_cJIcgM7rw&feature=youtu.be&t=52s

0.4 Datacamp: Python List Comprehension Tutorial

https://www.datacamp.com/community/tutorials/python-list-comprehension

[ ]:
```

```