MANIPAL
ACADEMY of HIGHER EDUCATION
(Institution of Eminence Deemed to be University)

MDS5103/ MSBA5104 Segment 04

LIST COMPREHENSION - TUTORIAL

Table of Contents

Introduction

Lists can be created by looping over a ‘for loop’ and adding elements into an empty list. Python provides effective and fast methods to create a list. This is referred to as list comprehension. This topic compares the creation of lists using traditional methods and list comprehension.

1. List Comprehension

List comprehensions are widely used in Python. List comprehension offers a short syntax to create a new list based on the values of an existing list. Unlike ‘for loop’, which requires multiple lines of code, list comprehension uses a single line of code.

1.1 Simple List Comprehension

Consider a list of elements. To create a new list in which each element is incremented by 1, a 'for loop’ can be used as shown in the code below. An empty list is initialised, and the existing list is iterated through. Each element in the list is incremented by 1 and the newly created elements are appended to the new list.

```
age = [23, 12, 67, 78, 84]
new_age = [ ]
for y in age:
        new_age.append(age+1)
print (new_age)
```

Output

[24, 13, 68, 79, 85]

The output displays the new list where each element has been incremented by 1.

The above method would be inefficient if the list was long. In such cases, list comprehension can be used. List comprehension produces the same result with a single line of code. The code below demonstrates the same. It is enclosed within **square brackets**, ‘y+1’ is the output expression, and ‘age’ is the iterable.

```
new_age = [ y +1 for y in age]
print (new_age)
```

Output

[24, 13, 68, 79, 85]

Python efficiently executes the code enclosed within square brackets and generates the new list as shown above. The output is the same as the one created using 'for loop'. Note that the list using list comprehension can be created over an iterable only and there must be an output expression. If needed a conditional statement using 'if' and 'else' can be used.

The code below demonstrates another example. In this, the squares of a list of numbers are calculated. In a traditional method, the list is iterated, and the square is calculated using the '**' operator for each number which is then appended to the result list.

```
#Suppose we want to create a list  containing the squares in the list  c
#Method  1
 c = [2, 3, 4, 5]
 s=[]
 for  num  in  c:
       s.append(num**2)
 print(s)
```

Output

[4, 9, 16, 25]

The code below demonstrates the calculation of squares using list comprehension. The code to compute the square of each number in a loop is enclosed within a square bracket. This is a concise and efficient way to compute the new list.

```
#Method  2
[x**2  for  x  in  c ] #Very  concise  way
```

Output

```
[4, 9, 16, 25]
```

The result in both cases are the squares of the numbers in the original list.

List comprehension can be applied on a string or a range object. The variable 'income' is of type integer in the code below. Using the 'income' variable within a list comprehension will result in an error as shown below.

```
Python_Structure = ['List', 'Strings','Dict','Boolean']
range(50)
Data_Science = 'Data is the new oil'
price = 'Rs.20,000'
income = 50000
[x+2 for x in income ]
```

Output

```
<ipython-input-77-6a3cbe895872> in <module>()
----> 1 [x+2 for x in income ]

TypeError: 'int' object is not iterable
```

The error occurs because integers are iterables.

1.2 List Comprehension Using Multiple Lists

Multiple lists can be combined to create pairs using list comprehension. The code below demonstrates the same. It contains a list of integers and a list of strings. The elements from the corresponding positions in the two lists should be combined to form a pair - [[1,'a'],[2,'b'],[3,'c'],[4,'d']]. This can be easily done using list comprehension as shown below.

```
a=[1,2,3,4]
b=['a','b','c','d']

[[x,y] for x in a for y in b if a.index(x)==b.index(y)]
```

Output

```
[[1, 'a'], [2, 'b'], [3, 'c'], [4, 'd']]
```

Note, using list comprehension, we can iterate through two 'for loops' in a single line of code. The code also uses an if condition to pair only those elements which are in the same index position.

The code below demonstrates a simple method to create pairs of numbers within two ranges. The 'for loop' is used to iterate over the range of numbers one within the other and a pair is appended to the list as shown below.

```
pairs = []
for p in range(0,2):
    for q in range(3,5):
        pairs.append([p,q])
print(pairs)
```

Output

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

However, this can be done more efficiently using list comprehension as shown below.

```
pairs2 = [[p,q] for p in range(0,2) for q in range(3,5)]
print(pairs2)
```

Output

[[0, 3], [0, 4], [1, 3], [1, 4]]

The output is the same in both the above cases. However, list comprehension reduces the amount of code significantly.

The 'zip()' function can be used to create a pair wise mapping based on the index of the elements in the list.

```
# zip
a=[1,2,3,4]
b=['a','b','c','d']
zip(a,b)
```

Output

<zip at 0x7fda255c4dc0>

As shown in the output above, the 'zip()' function creates a 'generator object' instead of an actual list. The content within the 'generator object' can be displayed by passing the 'generator object' as an argument to the 'list()' function as shown below.

```
print(list(zip(a,b)))
```

Output

[(1, 'a'), (2, 'b'), (3, 'c'), (4, 'd')]

The output is now a list of tuples. Each tuple contains elements in the same index positions from both lists.

To create a list of lists, the below code can be used. This uses list comprehension. 'list(x)' is the output expression and 'zip(a,b)' returns the iterable object.

```
[list(x) for x in zip(a,b) ]
```

Output

[[1, 'a'], [2, 'b'], [3, 'c'], [4, 'd']]

The output is now a list of lists.

1.3 Conditional Statements in List Comprehension

Conditional statements can also be used in list comprehension to filter out the elements in an 'array'. The code below demonstrates the same. The list 'x' consists of strings and integers. To remove or filter one type of element from 'x' and create a new list, we can use list comprehension with 'if condition'.

```
x = ['P', 6, 1, 'Q', 5, 'R']
#List Comprehension - Conditions on output expression
strings = [y for y in x if type(y) == str]
print(strings)
```

Output

['P', 'Q', 'R']

The 'if condition' checks for the type of element. The element is included in the output list only if the if condition returns 'True'.

The code below demonstrates another example, in this the squares of numbers are calculated only for even numbers. Thus, an 'if condition' can be used to check the remainder of the number after dividing it by 2 using the modulo operator, '%'. If the remainder is zero, then the number is even. The square of even numbers is then included in the list. This is done for all the numbers returned by 'range(10)' i.e., 0 to 9.

```
[num**2 for num in range(10) if num%2 ==0]
```

Output

[0, 4, 16, 36, 64]

Note the positions of the 'if' and 'for' statements can be interchanged. Also, an 'else condition' can be included. The code below demonstrates the same. In this, 0 is returned for all the odd numbers since the 'else' statement' returns 0.

```
[num**2 if num%2 ==0 else 0 for num in range(10)]
```

Output

```
[0, 0, 4, 0, 16, 0, 36, 0, 64, 0]
```

The output in this case is the square of the number if the number is even and 0 if the number is odd.

2. Generator Expression

List comprehension can return a list whereas generators return a generator object.

The generator expression looks like a list, but it does not store the list in the memory. It is an object that can be iterated over to produce elements of the list. It looks like a list except that **parentheses** are used instead of square brackets to create a generator. All other functionalities are the same as that of list comprehension.

The code shown below is similar to the examples used in a list comprehension. However, in this case, a parenthesis '()' is used instead of square brackets '[ ]'.

```
int_list = [-2, 4, 1, 6, -3]
print(x for x in int_list if x > 0)
print(list(x for x in int_list if x > 0))
```

Output

```
<generator object <genexpr> at 0x0000016FCB029AF0>
[4, 1, 6]
```

As shown in the output above, the result is a generator object and not the actual list. The generator object is useful in cases where the list is very large. It is efficient and saves a lot of memory space. To display all the elements from the generator object, it can be passed

as an argument to the 'list()' function as shown in the code above. This will display the list of elements [4,1,6].

E-references:

- Dive into Python by Mark Pilgrim

  Link: https://diveintopython3.net/

- Automate the Boring Stuff with Python by Al Sweigart

  Link: https://automatetheboringstuff.com/#toc