Comprehensions

Comprehensions in Python are a way to create a new sequence from an already existing sequence.

There are four main types of comprehensions in Python:

List comprehension

Dictionary comprehension

Set comprehension

Generator comprehension

You will now explore each of these to learn how to use them.

List comprehension

The syntax for list comprehension is:

`[ <expression> for x in <sequence> if <condition>]`

The same can be illustrated with the example below.

```
data = [2,3,5,7,11,13,17,19,23,29,31]

# Ex1: List comprehension: updating the same list
data = [x+3 for x in data]
print("Updating the list: ", data)

# Ex2: List comprehension: creating a different list with updated values
new_data = [x*2 for x in data]
print("Creating new list: ", new_data)

# Ex3: With an if-condition: Multiples of four:
fourx = [x for x in new_data if x%4 == 0 ]
print("Divisible by four", fourx)

# Ex4: Alternatively, we can update the list with the if condition as well
fourxsub = [x-1 for x in new_data if x%4 == 0 ]
print("Divisible by four minus one: ", fourxsub)

# Ex5: Using range function:
nines = [x for x in range(100) if x%9 == 0]
print("Nines: ", nines)
```

The output is:

```
Updating the list:  [5, 6, 8, 10, 14, 16, 20, 22, 26, 32, 34]
Creating new list:  [10, 12, 16, 20, 28, 32, 40, 44, 52, 64, 68]
Divisible by four [12, 16, 20, 28, 32, 40, 44, 52, 64, 68]
Divisible by four minus one:  [11, 15, 19, 27, 31, 39, 43, 51, 63, 67]
Nines:  [0, 9, 18, 27, 36, 45, 54, 63, 72, 81, 90, 99]
```

The given example provides different ways in which the list comprehensions can be used to update the list or generate a new list. Comprehensions provide a short-hand and elegant way of updating sequences. As may be evident, the same code can be written using the conventional for loop and if else conditions.
For instance, in the case of example 1:

```
# List comprehension:
data = [x+3 for x in data]

# Regular for loop:
for x in range(len(data)):
    data[x] = data[x] + 3
```

List comprehension can be a better option once you get the hang of it. It must be noted how the same concept can be extended to include multiple if else conditions as necessary.

List comprehensions are the most commonly used, but there are other types that can also make code pragmatic and simple. The structure and syntax for them are very similar to that of list comprehensions except for the data types that are used.

Dictionary comprehension

The syntax for dictionary comprehension is:

```
dict = { key:value for key, value in <sequence> if <condition> }
```

Dictionary comprehension takes one or two lists as input and creates a dictionary out of it. I will now demonstrate how this can be done using only one list and by using two lists.

```
# Using range() function and no input list
usingrange = {x:x*2 for x in range(12)}
print("Using range(): ",usingrange)

# Lists
months = ["Jan", "Feb", "Mar", "Apr", "May", "June", "July", "Aug", "Sept", "Oct", "Nov", "Dec"]
number = [1,2,3,4,5,6,7,8,9,10,11,12]

# Using one input list
numdict = {x:x**2 for x in number}
print("Using one input list to create dict: ", numdict)

# Using two input lists
months_dict = {key:value for (key, value) in zip(number, months)}
print("Using two lists: ", months_dict)
```

The output is:

```
Using range():  {0: 0, 1: 2, 2: 4, 3: 6, 4: 8, 5: 10, 6: 12, 7: 14, 8: 16, 9: 18, 10: 20, 11: 22}
Using one input list to create dict:  {1: 1, 2: 4, 3: 9, 4: 16, 5: 25, 6: 36, 7: 49, 8: 64, 9: 81,
Using two lists:  {1: 'Jan', 2: 'Feb', 3: 'Mar', 4: 'Apr', 5: 'May', 6: 'June', 7: 'July', 8: 'Aug
```

Note how in case of using two lists, the format it follows is:

new_dict ={key:value for (key, value) in zip(list1, list2)}

Here I used the zip function that combines the two lists. When the two lists are of unequal length, the length of the shorter list is the length of the dictionary.

Set comprehension

The set comprehension deals with the set data type and it's very similar to list comprehension. The only key difference is the use of curly brackets for sets instead of square brackets as in lists. For example:

```
set_a = {x for x in range(10,20) if x not in [12,14,16]}
print(set_a)
```

The output is:

```
{10, 11, 13, 15, 17, 18, 19}
```

You can see the code format is similar to what I used in list comprehensions. For the sake of showing versatility, I used the "not in" keywords to check the values in the list. The output is the values in ranges 10 and 20 that are not present in that list.

Generator comprehension

Generator comprehensions are also very similar to lists with the variation of using curved brackets instead of square brackets. They are also more memory efficient as compared to list comprehensions. For example:

```
data = [2,3,5,7,11,13,17,19,23,29,31]
gen_obj = (x for x in data)
print(gen_obj)
print(type(gen_obj))
for items in gen_obj:
    print(items, end = " ")
```

The output is:

```
<generator object <genexpr> at 0x102a87d60>
<class 'generator'>
2 3 5 7 11 13 17 19 23 29 31
```

In the code above, I created a generator object of the class generator instead of a list. The elements in this iterator object cannot be directly accessed and need the help of a for loop and as such, I iterate over these elements and print them.
We will shortly be looking at the difference between map() function and list comprehensions. Assuming we know there is some function called square() that exists as below:

```
def square(num):
    return num * 2
```

Here is the difference between map() function and list comprehensions:

```
newdata = map(square, data)
```

```
newdata = [x + 3 for x in data]
```

Notice how both map() functions and list comprehension effectively do the same job of modifying iterator sequences such as the list in the example above.
List comprehensions have been a relatively recent development but it does not necessarily mean they are more efficient. Comprehensions have gained popularity primarily for providing cleaner code readability and ease of use. They also provide some added advantages such as providing filtering using if else conditions.
List comprehensions also provide direct return of a list as compared to map() function that returns a map object. It is mainly the clarity that has made list comprehensions popular, but map() functions are still arguably a better choice when it comes to the use of larger sequences.