1) . What is the difference between enclosing a list comprehension in square brackets and parentheses?

Ans: Enclosing a list comprehension in square brackets returns a list.but where as enclosing a list comprehension in parentheses returns a `generator` object

In [1]:
```
l = [ele for ele in range(10)]
print(l, type(l))
g = (ele for ele in range(10))
print(g, type(g))
```

```
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9] <class 'list'>
<generator object <genexpr> at 0x000001FF01E5D510> <class 'generator'>
```

2) What is the relationship between generators and iterators?

Ans: An `iterator` is an object which contains a countable number of values and it is used to iterate over iterable objects like list, tuples, sets, etc. Iterators are implemented using a class. It follows lazy evaluation where the evaluation of the expression will be on hold and stored in the memory until the item is called specifically which helps us to avoid repeated evaluation. As lazy evaluation is implemented, it requires only 1 memory location to process the value and when we are using a large dataset then, wastage of RAM space will be reduced the need to load the entire dataset at the same time will not be there.For an iterator: `iter()` keyword is used to create an iterator containing an iterable object. `next()` keyword is used to call the next element in the iterable object.

Similarly `Generators` are an another way of creating iterators in a simple way where it uses the keyword `yield` statement instead of `return` statement in a defined function.Generators are implemented using a function. Just as iterators, generators also follow `lazy evaluation` . Here, the yield function returns the data without affecting or exiting the function. It will return a sequence of data in an iterable format where we need to iterate over the sequence to use the data as they won't store the entire sequence in the memory.

In [2]:
```
# Example of iterartor
iter_str = iter(['iNeuron','Full','Stack','Data Science'])
print(type(iter_str))
print(next(iter_str))
print(next(iter_str))
print(next(iter_str))
print(next(iter_str))
print(iter_str) # After the iterable object is completed, to use them again we

# Example of Generator
def cube_numbers(in_num):
    for ele in range(in_num+1):
        yield ele**3

out_num = cube_numbers(4)
print(next(out_num))
print(next(out_num))
print(next(out_num))
print(next(out_num))
print(next(out_num))
```

```
<class 'list_iterator'>
iNeuron
Full
Stack
Data Science
<list_iterator object at 0x000001FF01EE00D0>
0
1
8
27
64
```

3) What are the signs that a function is a generator function?

Ans: A generator function uses a `yield` statement instead of a `return` statement. A generator function will always return a iterable object called generator. where as a normal function can return a `string/list/tuple/dict/NoneType` ... etc

4) What is the purpose of a yield statement?

Ans: The yield statement suspends function's execution and sends a value back to the caller, but retains enough state to enable function to resume where it is left off. When resumed, the function continues execution immediately after the last yield run. This allows its code to produce a series of values over time, rather than computing them at once and sending them back like a list.

5) What is the relationship between map calls and list comprehensions? Make a comparison and contrast between the two ?

Ans: The main differences between map calls and list comprehensiosn are:

1. List comprehension is more concise and easier to read as compared to map.
2. List comprehension allows filtering. In map, we have no such facility. For example, to print all odd numbers in range of 50, we can write `[n for n in range(50) if n%2 != 0]`. There is no alternate for it in map
3. List comprehension are used when a list of results is required as final output.but map only returns a map object. it needs to be explicitly coverted to desired datatype.
4. List comprehension is faster than map when we need to evaluate expressions that are too long or complicated to express
5. Map is faster in case of calling an already defined function on a set of values.

In [ ]: