# Programming with python™

By

**Rami Tailakh**

Senior Software Engineer and Data Science Practitioner

MSc in Applied Computing and Information Technology

## Day-4 Agenda

- Day-3 Quick Review
- Closures
- Generators
- Iterators
- List Comprehensions

## Day-3 Challenges-Review

1. Write a Python function that computes the multiplication of all the numbers in a list.
2. Write a Python Program to Display Fibonacci Sequence. For example: 0, 1, 1, 2, 3, 5, 8, 13 and so on...
3. Write a Python program to print Even Numbers in a List. Hint: use **filter** function and **lambda** expression

## Python Closure

- Closure in Python is to define a function inside of another
- To write nested functions
- Example:

```
>>> def outer_function(x):
        def inner_function():
            return x
        return inner_function
>>> test_function = outer_function(10)
>>> test_function()
```

## Python Iterator

- It is an object which can be iterated returning one object at a time
- To read items of a list one by one
- The iter() and next() functions are used explicitly
- A Python iterator saves resources; only one element is stored in the memory at a time

```
>>> iter_object = iter('Python')

>>> next(iter_object) # Repeat the execution of this line
```

## Python Generator

- It generates a sequence of values like lists and tuples
- It is a simple way of creating iterators
- It is a kind of iterable over once
- Generators do not store all the values in memory, rather, they generate the values on the fly
- Generators can only be used once
- It is **concerning iteration**

## Python Generator-Continue

- To create a python generator, the **yield** statement is used inside a function
- The **yield** statement replaces the **return** of a function
- The **yield** statement suspends function’s execution and sends a value back to caller
- It produces a series of values over time, instead of computing them at once and sending them back as a list

## Python Generator-Continue

- Example:

```
>>> def counter():
        i=1
        while(i<=3):
            yield i
            i+=1
>>> for i in counter():
         print(i)
```

```
Out:    1

        2

        3
```

## List Comprehensions

- List Comprehensions make code more elegant
- A list Comprehension in Python allows create a new list
- A created list can be assigned to a variable
- It can be done by typing an expression followed by a for statement inside brackets
- An if-statement can be added to filter out items (Optional)

## List Comprehensions-Continue

- List Comprehension in Python can be summarised as follows:

**<variable> = [<expression>   <iterator>   <filtration>]**

```
>>> even_numbers=[i for i in range(1,11) if i%2==0]

>>> print(even_numbers)
```

Out: **[2, 4, 6, 8, 10]**

## List Comprehensions-Continue

- Interestingly, Python list comprehension allows do coding in one line
- Here is an example of splitting a string into characters

| For loop | Comprehensions |
|---|---|
| `>>> characters_list=[]`<br>`>>> for i in 'Python':`<br>`        characters_list.append(i)`<br>`>>> print(characters_list)` | `>>> characters_list = [c for c in 'Python']`<br>`>>> print(characters_list)` |

- Try this: list('Python')

## List Comprehensions-Example-1

Extract the words in a text that are more the 5 letters.

```
>>> string = 'This is Python. Python is very powerful.'

>>> print([w for w in string.replace('.','').split(' ') if len(w)>5])
```

Out: **['Python', 'Python', 'powerful']**

## List Comprehensions-Example-2

Extract numerical tokens in a text string

```
>>> def is_number(w):
        try:
            w = int(w)
            return True
        except:
            return False

>>> text = 'Python 2 is deprecated. It is not supported any more after
December 2019'

>>> [w for w in text.replace('.', '').split(' ') if is_number(w)]
```

Out: ['2', '2019']

## Nested List Comprehensions

- It is how to use a Python list comprehension for a nested for-loop
- However, it does not makes sense to write a very long list comprehension
- Syntax:

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

## Nested List Comprehensions-Example-1

Print the multiplication number table of the numbers from 1-12.

```
>>> [['{}x{}={}'.format(i,j,i*j) for j in range(1,13)] for i in range(1,13)]
```

## Nested List Comprehensions-Example-2

Find all letters used in a string

```
>>> words = ["apples", "bird", "cat", "dog", "elephant",
"fox","hen",'jaguar','whale','monkey', 'quiz','van']

>>> set([letter for word in words

                for letter in word])
```

Out: **<Was it all English alphabets? Was it sorted?>**

## Nested List Comprehensions-Practice

Find the occurrences of each letter used in the list of words used in the previous example.

```
>>> [<What should be here> for word in words
                            for letter in word]
```

## Dictionary Comprehensions

- Transforming one dictionary into another dictionary
- Being able to access the key and the value objects of a dictionary

```
>>> {k: v for k, v in my_dictionary.items()}
```

## Dictionary Comprehensions-Example

Find the averages and saves the results into another dictionary

```
>>> student_marks = {'Samer': [90, 88, 82],
                     'Adam': [80, 98, 79],
                     'Mo': [80, 85, 90]}


>>> student_averages = {k: sum(v)/3 for k, v in student_marks.items()}
>>> student_averages
```

## Challenges

1. Write a function that takes a list of numbers and returns a list of even numbers only. The function should be one line of Python code.
2. Write a module that implements the Caesar cipher.
3. Write a function that extracts special characters from a text.
4. Write a Python code that finds the most (3) frequent words in a text.
5. Fibonacci Sequence. Again! But do not use a recursive function.