Iterables

Iterables are a kind of entity which we are able to itearte by using some kind of function and then we make that a iterator to extract dataset one by one. This is usually done using a for-loop. Objects like lists, tuples, sets, dictionaries, strings, etc. are called iterables. In short, anything you can loop over is an iterable.

In [ ]:

```
for i in 'shivam':
    print(i)
```

```
s
h
i
v
a
m
```

In [ ]:

```
for x in ['Python', 'Java', 'C', 'C++', 'Ruby']:
    print(x)
```

```
Python
Java
C
C++
Ruby
```

Iterators

An Iterator is an object that produces the next value in a sequence when you call next(object) on some object. Moreover, any object with a next method is an iterator. An iterator raises StopIteration after exhausting the iterator and cannot be re-used at this point.

In [ ]:

```
a = 'Python'
```

In [ ]:

```
next(a)
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-4-15841f3f11d4> in <module>
----> 1 next(a)

TypeError: 'str' object is not an iterator
```

In [ ]:

```
# now we can make this an iterator by using an iter function

a = iter('Python')
```

In [ ]:

```
next(a)
```

Out[ ]:

```
'P'
```

In [ ]:

```
l = [2,3,'Shivam', 7, 9]
it = iter(l)
```

In [ ]:

```
next(it)
```

Out[ ]:

```
2
```

In [ ]:

```
next(it)
```

Out[ ]:

```
3
```

In [ ]:

```
next(it)
next(it)
next(it)
```

Out[ ]:

```
9
```

In [ ]:

```
next(it)
```

```
---------------------------------------------------------------------------
StopIteration                             Traceback (most recent call last)
<ipython-input-12-bc1ab118995a> in <module>
----> 1 next(it)

StopIteration:
```

That's right, we get an error! If we try to access the next value after reaching the end of an iterable, a StopIteration exception will be raised which simply says "you can't go further!".

Generator

***Generators* are also iterators but are much more elegant. It is a kind of object which keep on generating new dataset remembering the fact that what i am generating last time.**

In [ ]:

```
def test(n):
    for i in range(n):
        yield i**3
```

The yield keyword works like a normal return keyword but with additional functionality – it remembers the state of the function. So the next time the generator function is called, it doesn't start from scratch but from where it was left-off in the last call.

In [ ]:

```
test(5)
```

Out[ ]:

```
<generator object test at 0x000001527BAB7148>
```

In [ ]:

```
a = test(5)
```

In [ ]:

```
next(a)
```

Out[ ]:

```
0
```

In [ ]:

```
next(a)
next(a)
next(a)
next(a)
```

Out[ ]:

```
64
```

In [ ]:

```
next(a)
```

```
---------------------------------------------------------------------------
StopIteration                             Traceback (most recent call last)
<ipython-input-24-15841f3f11d4> in <module>
----> 1 next(a)

StopIteration:
```

In [ ]:

```
# Program to display the Fibonacci sequence with generator function

def genfi(n):
    a = 1
    b = 1

    for i in range(n):
        yield a
        a, b = b, a+b
```

In [ ]:

```
# Creating a generator object

g = genfi(5)
```

In [ ]:

```
next(g)
next(g)
next(g)
next(g)
next(g)
```

Out[ ]:

```
5
```

List Comprehension

List comprehensions in Python are concise, syntactic constructs. They can be utilized to generate lists from other lists by applying functions to each element in the list. A list comprehension creates a new list by applying an expression to each element of an iterable.

The Basic syntax of List Comprehension are:

In [ ]:

```
# To create a list of squared integers:

squares = [x * x for x in (1, 2, 3, 4, 5)]
squares
```

Out[ ]:

```
[1, 4, 9, 16, 25]
```

In [ ]:

```
# Get a list of uppercase characters from a string

[s.upper() for s in "Hello World"]
```

Out[ ]:

```
['H', 'E', 'L', 'L', 'O', ' ', 'W', 'O', 'R', 'L', 'D']
```

Conditional List Comprehensions

Given a list comprehension you can append one or more if conditions to filter values. The basic syntax are:

In [ ]:

```
# list of even numbers

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

Out[ ]:

```
[0, 2, 4, 6, 8]
```

else

else can be used in List comprehension constructs, but be careful regarding the syntax. The if/else clauses should be used before for loop, not after:

In [ ]:

```
# create a list of characters in apple, replacing non vowels with '*'
# When using if/else together use them before the loop

[x if x in 'aeiou' else '*' for x in 'apple']
```

Out[ ]:

```
['a', '*', '*', '*', 'e']
```

In [ ]:

```
# One can combine ternary expressions and if conditions. The ternary operator works on th
e filtered result:

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

Out[ ]:

```
['*', '*', 4, 6, 8]
```

Dictionary Comprehensions

A dictionary comprehension is similar to a list comprehension except that it produces a dictionary object instead

of a list.

In [ ]:

```
d = {i: i**2 for i in range(11)}
d
```

Out[ ]:

```
{0: 0, 1: 1, 2: 4, 3: 9, 4: 16, 5: 25, 6: 36, 7: 49, 8: 64, 9: 81, 10: 100}
```

As with a list comprehension, we can use a conditional statement inside the dict comprehension to produce only the dict elements meeting some criterion.

In [ ]:

```
nd = {name: len(name) for name in ('Python', 'Training', 'Program') if len(name) > 6}
```

In [ ]:

```
nd
```

Out[ ]:

```
{'Training': 8, 'Program': 7}
```

Merging Dictionaries

Combine dictionaries and optionally override old values with a nested dictionary comprehension.

In [ ]:

```
dict1 = {'w': 1, 'x': 1}
dict2 = {'x': 2, 'y': 2, 'z': 2}

{k: v for d in [dict1, dict2] for k, v in d.items()}
```

Out[ ]:

```
{'w': 1, 'x': 2, 'y': 2, 'z': 2}
```

List Comprehensions with Nested Loops

List Comprehensions can use nested for loops. You can code any number of nested for loops within a list comprehension, and each for loop may have an optional associated if test. When doing so, the order of the for constructs is the same order as when writing a series of nested for statements.

In [ ]:

```
#List Comprehension with nested loop

[x + y for x in [1, 2, 3] for y in [3, 4, 5]]
```

Out[ ]:

```
[4, 5, 6, 5, 6, 7, 6, 7, 8]
```

In [ ]:

```
# print the list of tables from 1 to 10.

t = [[i*j for j in range(1,11)] for i in range(2,11)]
t
```

Out[ ]:

```
[[2, 4, 6, 8, 10, 12, 14, 16, 18, 20],
 [3, 6, 9, 12, 15, 18, 21, 24, 27, 30],
 [4, 8, 12, 16, 20, 24, 28, 32, 36, 40],
```

```
 [5, 10, 15, 20, 25, 30, 35, 40, 45, 50],
 [6, 12, 18, 24, 30, 36, 42, 48, 54, 60],
 [7, 14, 21, 28, 35, 42, 49, 56, 63, 70],
 [8, 16, 24, 32, 40, 48, 56, 64, 72, 80],
 [9, 18, 27, 36, 45, 54, 63, 72, 81, 90],
 [10, 20, 30, 40, 50, 60, 70, 80, 90, 100]]
```

Set Comprehensions

Set comprehension is similar to list and dictionary comprehension, but it produces a set, which is an unordered collection of unique elements.

In [ ]:

```
# A set of even numbers between 1 and 10:

e = {x for x in range(1, 15) if x % 2 == 0}
e
```

Out[ ]:

```
{2, 4, 6, 8, 10, 12, 14}
```

Iterate two or more list simultaneously within list comprehension

For iterating more than two lists simultaneously within list comprehension, one may use zip() as:

In [ ]:

```
list_1 = [1, 2, 3 , 4]
list_2 = ['a', 'b', 'c', 'd']
list_3 = ['6', '7', '8', '9']
```

In [ ]:

```
# Two lists

[(i, j) for i, j in zip(list_1, list_2)]
```

Out[ ]:

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

In [ ]:

```
# Three lists

[(i, j, k) for i, j, k in zip(list_1, list_2, list_3)]
[(1, 'a', '6'), (2, 'b', '7'), (3, 'c', '8'), (4, 'd', '9')]
```

Out[ ]:

```
[(1, 'a', '6'), (2, 'b', '7'), (3, 'c', '8'), (4, 'd', '9')]
```

In [ ]:

```

```

In [ ]:

```

```