# Geneartor:-

- Use generator when you deal with stream of data — possibly *infinite stream*. On the other hand if you need to iterate over and over on a set of values leverage the *list*.
- Know the difference between **generator expression** and **generator function**
- *Howto* write a **generator function** is here
- *Howto* pipe many streams is here
- *Howto* iterate over all the records of all files in all folders for a given path is here
- *Howto* dump a database to S3 using **generators** is here
- *Howto* consume a Kinesis stream is here

# Generator or list?

Let's look at the following code snippet:
`

```
# List
comprehension
            l = [i for i in range(10)]
            # Generator expression
            g = (i for i in range(10))
```

The difference between the 2 expressions seems insignificant: one uses **brackets** and the other one uses **parenthesis**. However the difference is major: the first one is a *list comprehension* and the second one is a *generator expression* and these are 2 distincts object in Python.

Let's start with the bracket version; At evaluation time a *list comprehension* is simply a list; it is available in memory and any list operations can be perform on it.

On the other hand a *generator expression* evaluates to a *generator* and that is a different animal than a *list*. The only common point between a *generator* and a list is that both of them can be iterated over. The major difference is that *generator expression* has a lazy behaviour; meaning elements of a generator expression are not available in memory, they are computed when requested: on an *on-demand* basis.

To put it another way, let's first review the type of operations available on a Python *list*; then let's make the comparison with *generator* objects. I group the operations that can be performed on a *list* in 3 categories: (a) index operations, (b) mutation operations and (c) iteration operation. We will see that for a *generator* a only the iteration operation is available.

## Index operations on list

The main operations are accessing *list* elements by *index*, or *slicing* the list. Note that the **len** method is put in this category because the computation of the length of a list is equivalent to returning its maximum *index*. So these are mainly read-only operations perform on the list object, e.g.

```
my_list =
[i for i
in
range(10)]
          my_list     # return [0,1,2,3,4,5,6,7,8,9]
          len(my_list)      # return 10, the length of l
          my_list[3]        # return 2
          my_list[1:5]      # return the new list [1, 2, 3, 4]
          my_list[:]        # return a new list copy of my_list
          my_list.count(x)  # return the number of times x appears in
          my_list
```

```
my_list.index(x)  # return the index of x the first time it
occurs in my_list
my_list.copy()    # return a new list copy of my_list
```

## Mutation operations on list

These are operations that change a list. Note that we are talking about in place modification as opposed to making a new copy of the list.

```
my_list =
[i for i
in
range(10)]
          my_list    # return [0,1,2,3,4,5,6,7,8,9]
          # mutation operations
          my_list.reverse()        # my_list is [9,8,7,6,5,4,3,2,1,0]
          my_list.sort()           # my_list is [0,1,2,3,4,5,6,7,8,9]
          my_list.append(10)       # my_list is [0,1,2,3,4,5,6,7,8,9,10]
          my_list.extend([11,12])  # my_list is
          [0,1,2,3,4,5,6,7,8,9,10,11,12]
          my_list.insert(3, 99)    # my_list is
          [0,1,2,99,3,4,5,6,7,8,9,10,11,12]
          ...
```

## Iteration on list

Iterating over the element of a list is probably the most common operation in Python. The main point to note here is that a list can be iterated over and over again, e.g.

```
my_list =
[i for i
in
range(10)]
          # iterate through the elements of my_list
          for el in my_list:
              print(el)
```

```
#... I can iterate as many time as I want
for el in my_list:
    print(el)
```

## So…, what is a generator?

We have just seen 3 types of operation allowed on a *list* and it is important to note that 2 of them (*indexing* and *mutation*) are possible only because list elements are available in memory. Since a *generator* is *lazy* — meaning its elements are computed *on-demand* — *indexing* and *mutation* cannot be performed. Only *iteration* can be performed on a *generator* object, and even then iteration can be performed only once.

So, you might ask what is the point of using a *generator* over a list, if the only operation allowed is iteration?

The answer is the usage depends of the use-case. I like to think about *generator* as an abstraction for *stream*. I use *generators* every time that I deal with stream of data — possibly *infinite stream*. If you think about it it makes sense, because element in stream does not need to be indexed and it does not make sense to mutate a stream… unless you mean on the fly operations, i.e. on an *on-demand basis*. In conclusion of this section:

Use generator when you deal with stream of data — possibly infinite. On the other hand if you need to iterate over and over on a set of values leverage the list.

In the following sections we will demonstrate some use-case of generators.

# Generators examples

## E0. Introducing Yield

You are already familiar with generator expression:

```
my_gen
= (i
for i
in
['a',
'b',
'c'])
      for el in my_gen:
          print(el, )  # prints a, b and c
```

Here is another way to write it using generator function:

```
def
gen_func():
                yield 'a'
                yield 'b'
                yield 'c'
            my_gen = gen_func()
            for el in my_gen:
              print(el, )   # prints a, b and c
```

Note the use of the keyword **yield** in *gen_func*;
the **yield** keyword is what makes a function a *generator function*. **Yield** statement is similar to **return** statement in the sense that both return a computed value.

The difference is that after a **return** a function will exit and that's it. On the other hand, after a **yield** the function does not exit, instead it looks for the next **yield** and wait there for someone to ask for it; in other words the function yields computed value on an *on-demand* basis. And the method to ask for a value to a *generator* function is by *iterating* over it.

## E1. An infinite stream of numbers

Here is a simple example of an infinite numbers generator:

```
def
gen_numbers():
                    n = 0
                    while True:
                        yield n
                        n += 1
```

Now let's pipe our infinite stream into different streams each one performing a specific operations, e.g filtering, maths, etc. For the purpose of the demonstration we've added a **break** condition in the previous **while** loop; otherwise the script below will run indefinitely... remember, it's an infinite number generator!

```
def
gen_numbers():
                        n = 0
                        while True:
                            yield n
                            n += 1
                            if n == 10000:break
                    # let's pipe our stream into other
                    streams...
                    s1 = (n for n in gen_numbers() if n%2 == 0)
                    # only even numbers
                    s2 = (n**2 for n in s1)
                    # squared the even numbers
```

```
s3 = (n for n in s2 if n < 10)
# only the first 10 of squared even numbers
for el in s3:
  print(el, )  # prints 0, 4
```

Note how easy it is to modify our initial stream; note also that all operations performed aren't evaluated until the last for loop, when we actually fetch elements from the final stream.

There is several way to write the script above, it is a matter of taste. Let's rewrite it using generator function only:

```
def
gen_numbers():
                    n = 0
                    while True:
                        yield n
                        n += 1
                        if n == 10000:break

                def gen_even():
                    for el in gen_numbers():
                        if el%2 == 0: yield el
                def gen_squared():
                    for el in gen_even():
                        yield el ** 2
                def gen_trunc():
                    for el in gen_squared():
                        if el < 10: yield el

                for el in gen_trunc():
                    print(el, )
```

Note how each generator builds upon the previous one. As always no computation is performed until the last **for loop** when the elements are fetched.