List Generators Cheat Sheet

1. Definition

- instead of [ ] use ( )
- generators do not contruct lists nor store them in memory
- However they are elements that can be iterated over to produce elements of the list as required.
- The main advantage of generator over a list is that it takes much less memory. We can check how much memory is taken by both types using sys.getsizeof() method.

In [1]:
```
list2 = (x for x in range(10))
list3 = (x for x in range(10))
```

- The list is generated when it is needed as follows

In [4]:
```
for x in list2:
    print(x)
```

```
0
1
2
3
4
5
6
7
8
9
```

In [7]:
```
print(next(list3))
print(next(list3))
```

```
0
1
```

I think

- You cannot run the for loop and the next() funtion without re-generating the geenrator
- Because the generator doesn't actually construct a list once you run a for loop through all its values there will be nothing left
- in which case you will get
- StopIteration exception is raised when there are no elements left to call.

In [8]:
```
# Create generator object: result
result = (num for num in range(0,31))

# Print the first 5 (0-4) values
print(next(result))
print(next(result))
print(next(result))
print(next(result))
print(next(result), '\n')

# Print the rest(5-30)  of the values. you can see that is starts from
for value in result:
    print(value)
```

```
0
1
2
3
4

5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29
30
```

In [9]:
```
list4 = (digits for digits in range(10))

gen_list = list(list4)
print(gen_list)
```

```
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

2. Memory

- compare the following
- the comprehension takes forever to compute (i literally hear my pc fan getting louder)
- while the generator is created instantly

In [12]:
```
# DO NOT RUN THIS. LEAVE AS COMMENT iF YOU DO YOUR PC WILL FREEZE

# count = [num for num in range(10 ** 1000000)]
# print(count)
```

In [10]:
```
count_gen = (num for num in range(10 ** 1000000))
print(next(count_gen))
print(next(count_gen))
```

```
0
1
```

3. Same Rules as Constructors

In [20]:
```
even = (num for num in range(1,10) if num % 2 == 0)
print(list(even))
```

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

In [21]:
```
# Create a list of strings: lannister
lannister = ['cersei', 'jaime', 'tywin', 'tyrion', 'joffrey']
```

In [22]:
```
# Create a generator object: lengths
lengths = (len(person) for person in lannister)

# Iterate over and print the values in lengths
for value in lengths:
    print(value)
```

```
6
5
5
6
7
```

---

4. Generator Functions

- They are defined like regular functions with def:
- The dont use keyword **return** they use **yield**
- They yield sequence of values instead of returning a single value

In [16]:
```
def num_sequence(n):
    """Generates values from 0 to n"""
    i = 0
    while n > i:
        yield i
        i += 1
```

In [17]:
```
print(num_sequence(5))
```

```
<generator object num_sequence at 0x10861b150>
```

for item in num_sequence(6):

```
print(item)
```

In [4]:
```
# Create a list of strings: lannister
lannister = ['cersei', 'jaime', 'tywin', 'tyrion', 'joffrey']
```

In [5]:
```
# Define generator function get_lengths
def get_lengths(input_list):
    """Generator function that yields the length of the strings in inp

    # Yield the length of a string
    for person in input_list:
        yield len(person)
```

In [6]:
```
# Print the values generated by get_lengths()
for value in get_lengths(lannister):
    print(value)
```

```
6
5
5
6
7
```