INTERMEDIATE PYTHON

LESSON 7 |Generators| 16-01-19

***Video tutorial*:**

Python Generators: ***https://www.youtube.com/watch?v=nvaIaz3F3K8***

Python generators

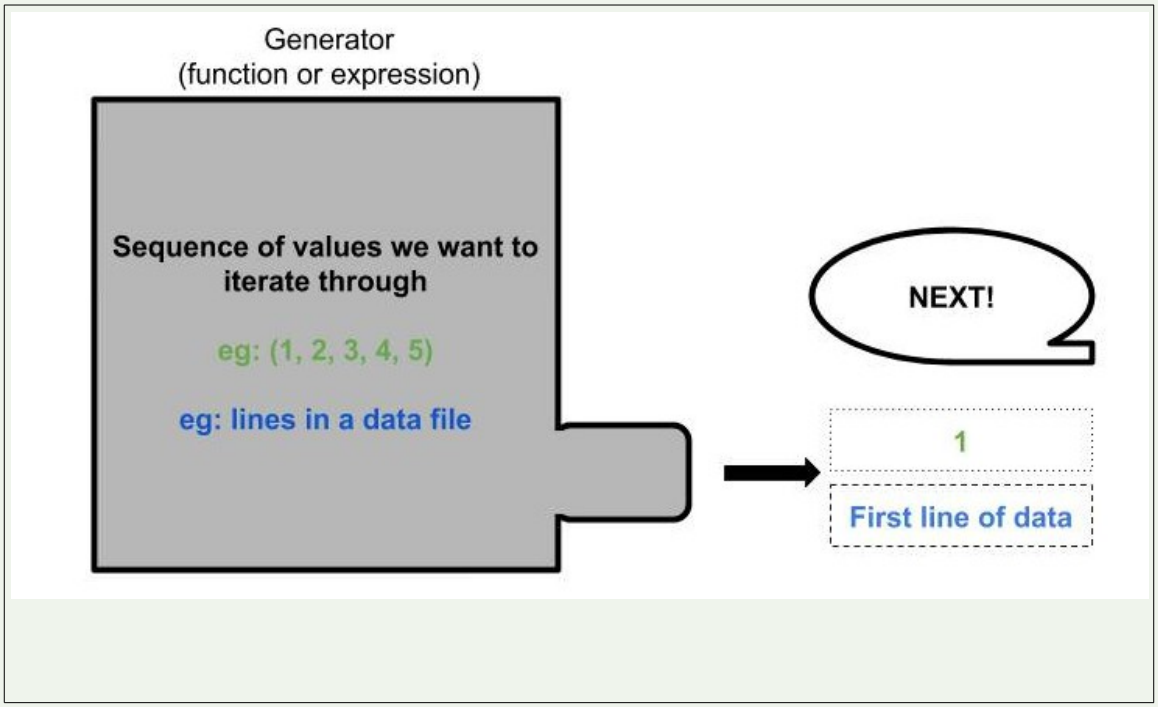
Generators

Generators are functions that can be **paused** and **resumed** on the fly, returning an object that can be iterated over. Unlike lists, they are lazy and thus **produce items one at a time and only when asked.** So they are much more **memory efficient** when dealing with large datasets.

To create a generator, you would define a function as you normally but use the **yield** statement instead of **return**, indicating to the interpreter that this function should be treated as an **iterator.** The **yield** statement pauses the function and saves the local state so that it can be resumed right where it left off.

How is that possible? How can a function be paused and then resumed with its local state kept intact? For all that we know, functions have a **single entry point and multiple exit points** (return statements). Each time we call a function, the code executes beginning from the first line of the function until it encounters an exit point. At that juncture, control is returned to the caller of the function and the function's stack of local variables is cleared and the associated memory reclaimed by the OS. Generator functions however don't behave this way. **They have multiple entry and exit points** . Each yield statement in a generator function simultaneously defines an exit point and a re-entry point. ***Execution of a generator function continues until a yield statement is encountered.*** *At that point, the local state of the function is preserved and the flow of control is yielded to the caller of the generator function*

It's easy to think of generators as a machine that waits for one command and one command only: `next()`. Once you call `next()` on the generator, it will dispense the next value in the sequence it is holding. Otherwise, you can't do much else with a generator



Example

```
def yrange(n):
    i = 0
    while i< n:
        yield i
        i+=1
```

```
>>> y = yrange(3)
>>> y
<generator object yrange at 0x038BFD70>
>>> next(y)
0
>>> next(y)
1
>>> next(y)
2
>>> next(y)
Traceback (most recent call last):
  File "<pyshell#5>", line 1, in <module>
    next(y)
StopIteration
```

Each time the yield statement is executed the function generates a new value.

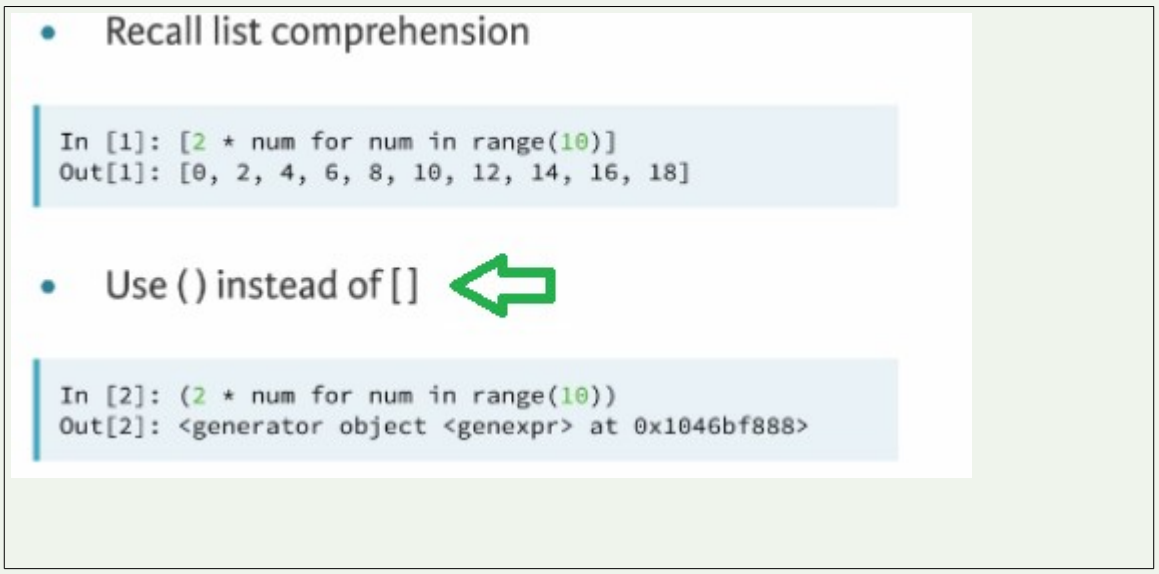
The **next()** method is called on the iterator object to get the next element of the sequence.

A **StopIteration** exception is raised when there are no elements left to call,

Generator Expressions

Generator Expressions are generator version of list comprehensions. They look like list comprehensions, but returns a generator back instead of a list.

We can use the generator expressions as arguments to various functions that consume iterators.

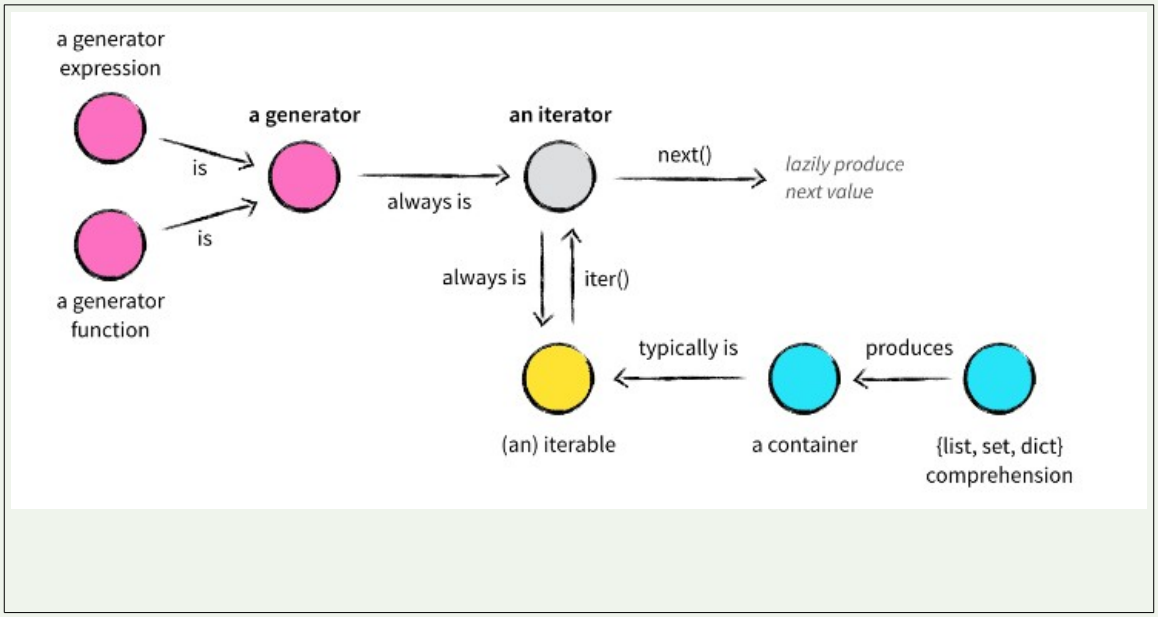


Example:

```
gen_exp = (x ** 2 for x in range(10) if x % 2 == 0)
```

```
>>> y = gen_exp
>>> y
<generator object <genexpr> at 0x02AFFD70>
>>> next(y)
0
>>> next(y)
4
>>> next(y)
16
>>> next(y)
36
>>> next(y)
64
>>> next(y)
Traceback (most recent call last):
  File "<pyshell#7>", line 1, in <module>
    next(y)
StopIteration
```

Summary


a generator expression
is
a generator
an iterator
next()
lazily produce next value
always is
is
a generator function
always is
iter()
typically is
produces
(an) iterable
a container
{list, set, dict} comprehension

Exercices:(<u>use generator function/expression</u>)

1. Write a program that takes a list of filenames as arguments and prints contents of all those files,

```
def readfiles(filenames):
    for f in filenames:
        for line in open(f):
            yield line

def grep(pattern, lines):
    return (line for line in lines if pattern in line)

def printlines(lines):
    for line in lines:
        print line,

def main(pattern, filenames):
    lines = readfiles(filenames)
    lines = grep(pattern, lines)
    printlines(lines)
```

2. Write a program that takes one or more filenames as arguments and prints all the lines which are longer than 40 characters.
3. Write a function that recursively descends the directory tree for the specified directory and generates paths of all the files in the tree.
4. Write a function to compute the number of python files (.py extension) in a specified directory recursively
5. Write a function to compute the total number of lines of code in all python files in the specified directory recursively.
6. Write a function to compute the total number of lines of code, ignoring empty and comment lines, in all python files in the specified directory recursively.

7. Write a program that takes an integer n and a filename as command line arguments and splits the file into multiple small files with each having n lines.