Generators

Here is a problem.

Write a **function** f with the following behavior: each time f is called it returns the next positive even integer. So, the first time it is called it returns 2, the next time 4 and so on.

Answer:

Problem

Now do it without global variables.

Generator functions are one of Python's most interesting and powerful features. Generators are often presented as a convenient way to define new kinds of iteration patterns. However, there is much more to them: Generators can also fundamentally change the whole execution model of functions. applications of generators.

Generators in Python are like factories that produce a sequence of values, one at a time.

They are created using a special type of function that uses the **yield** keyword to produce values on-the-fly.

When a **generator function** is called, it returns a **generator object** that can be iterated over, just like a list, but it generates values only when requested. This makes generators very efficient, as they generate values only when needed, and they can be used to generate large sequences of values that cannot fit into memory at once. A simple example of a generator function is one that generates a sequence of even numbers, and another example is one that generates the Fibonacci sequence.

What is the difference between a generator function and a generator object?

In Python, a generator is a type of function that produces a generator object. A generator object is the actual iterable object that is returned by the generator function when it is called.
The generator function defines the logic for generating values on-the-fly using the **yield** keyword. When the generator function is called, it returns a generator object which can be iterated over using a loop or a comprehension. The generator object keeps track of the state of the generator function and generates values on-the-fly as they are requested by the program.

Therefore, the main difference between a generator and a generator object is that the generator is the function itself, while the generator object is the iterable object that is produced by the generator function. The generator function is the blueprint for the generator object, defining how values should be generated and returned, while the generator object is the actual instance of the generator, used to iterate over and generate the values.

For example:

```
def even_numbers(n):
   i = 0
   while i < n:
      yield i
      i += 2
```

The function above is a generator function, it uses the “yield” keyword.

```
even_gen = even_numbers(10)
```

even_gen is a generator object and it can be iterated over like any other iterable.

```
for num in even_gen:
   print(num)
```

Another example:

```
def countdown(n):
    print('Counting down from', n)
    while n > 0:
        yield n
        n -= 1

for x in countdown(10):
   print('T-minus', x)
```

If you call this function, you will find that none of its code starts executing. For example:

```
>>> c = countdown(10)
>>> c
<generator object countdown at 0x105f73740
> >>>
```

Instead, a generator object is created. The generator object, in turn, only executes the function when you start iterating on it.

next

We saw that we can do this with the for loop. Another way is to call next() on it:

```
>>> next(c)
Counting down from 10
10
>>> next(c)
 9
```

When next() is called, the generator function executes statements until it reaches a yield statement. The yield statement returns a result, at which point execution of the function is suspended until next() is invoked again. While it's suspended, the function retains all of its local variables and execution environment. When resumed, execution continues with the statement following the yield.

next() is a shorthand for invoking the __next__() method on a generator. For example, you could also do this:

```
>>> c.__next__()
8
>>> c.__next__()
7
>>>
```

Problem:

Write a generator that will yield the "next" prime number each time that it's called.

Problem:

Write a generator that will yield the "next" leap year each time that it's called.

Problem:

Using a generator, create a dictionary such that d[i] is mapped to the i^{th} prime number.

Can you also do this with a dictionary comprehension?

Problem:

Write a generator get_oct() that will produce the next octal number each time it's called. The value should be a string, so that when the octal number 20 is produced (this is the octal representation for the decimal 16) the generator will return '20'.

Problem:

Write a generator that produces "serial numbers" according to the following rule: a serial number is a ten-character **string** where

- the first two characters are upper-case alphabetic and
- the following eight are numeric.

The alphabetic portion will be all strings from AA through ZZ in lexicographic order (AA,AB,AC …). For **each** alphabetic prefix, the numeric part will be all strings 00000000 – 99999999. So we have AA00000000, AA00000001 …ZZ99999999.

Answer

Problem:

Write a generator that monitors a device that periodically sends either a 1 or a 0. The generator keeps track of the number of zeros between receiving a 1. They call that number the gap. Each time a value is received the generator yields the gap.

For example, say the received sequence is:

runs=[1,1,0,1,1,0,0,0,1,1,1,1,0,1,0,0,1]

then the generator will output:

[0, 0, 1, 0, 3, 0, 0, 0, 1, 2, 0]

Write the generator.

Problem:

Referring to the problem above. Imagine that the sequence of 1s and 0s represents the state of some security sensor, 1 if available, zero if not. If the gap between 0s exceeds some user defined n, the generator should return a warning and return. Write the modified generator.

A generator function produces items until it "returns" (since it's a function after all)—by

- **reaching the end of the function or**
- **by using a return statement**.

Question:

What happens when we run this:

```
def even_numbers1():
   i = 0
   yield i
   i += 2

even=even_numbers1()


for w in range(10):
   print(next(even))
```

Why?

This raises a StopIteration exception that terminates a for loop.

If a generator function returns a non-None value, it is **attached** to the StopIteration exception.

For example, this generator function uses both yield and return:

```
def func():
    yield 37
    return 42
```

Here's how the code would execute

```
>>> f = func()
>>> f
<generator object func at 0x10b7cd480>
>>> next(f)
37
>>> next(f)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
StopIteration: 42
>>>
```

Notice that the return value is "attached" to StopIteration. To collect this value, you need to explicitly catch StopIteration and extract the value:

```
try:
    next(f)
except StopIteration as e:
    value = e.value
```

Normally, generator functions don't return a value. Generators are almost always consumed by a for loop where there is no way to obtain the exception value. This means the only practical way to get the value is to drive the generator manually with explicit next() calls. Most code involving generators just doesn't do that.

A subtle issue with generators is where a generator function is only partially consumed. For example, consider this code that abandons a loop early:

```
for n in countdown(10):
    if n == 2:
        break
    statements
```

In this example, the for loop aborts by calling break and the associated generator never runs to full completion. If it's important for your generator function to perform some kind of cleanup action, make sure you use try-finally or a context manager. For example:

```
def countdown(n):
    print('Counting down from', n)
    try:
        while n > 0:
            yield n
            n = n - 1
    finally:
        print('Only made it to', n)
```

Generators are guaranteed to execute the finally block code even if the generator is not fully consumed—it will execute when the abandoned generator is garbage-collected. Similarly, any cleanup code involving a context manager is also guaranteed to execute when a generator terminates:

```
def func(filename):
    with open(filename) as file:
        ...
        yield data
        ...
    # file closed here even if generator is abandoned
```

Proper cleanup of resources is a tricky problem. As long as you use constructs such as try-finally or context managers, generators are guaranteed to do the right thing even if they are terminated early.

Restartable Generators (as a class) – we will see this later

Normally a generator function executes only once. For example:

```
>>> c = countdown(3)
>>> for n in c:
...    print('T-minus', n)
...
T-minus 3
T-minus 2
T-minus 1
>>> for n in c:
...    print('T-minus', n)
...
>>>
```

If you want an object that allows repeated iteration, define it as a class and make the __iter__() method a generator:

```
class countdown:
   def __init__(self, start):
      self.start = start

   def __iter__(self):
      n = self.start
      while n > 0:
         yield n
         n -= 1
```

This works because each time you iterate, a fresh generator is created by __iter__().