Generators

What is the difference between:

1. **print(sum([1,2,3,4,5]))**
2. **print(sum(range(5)))**
3. **print(sum(i for i in range(1,6)))**

since they all print 15?

Answer

Here is a problem.

Write a **function** f with the following behavior: each time f is called it returns the next positive even integer less than 10. 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.

Review of iterators: What is an iterable and what is an iterator?

The terms "iterable" and "iterator" in Python are closely related but serve different roles in the process of iteration:

Definition: An **iterable** is any Python object that can return its members one at a time, permitting it to be iterated over in a loop. Common examples include lists, tuples, sets, dictionaries, strings, and even file objects. We will later when we build our own classes that we can create iterables as well.

You typically interact with iterables when you use loops (like a `for` loop) or other functions that expect something they can iterate over (like `sum()`, etc.).

Definition: The iterable implements its "one at a time" behavior via an associated iterator. An **iterator** is an object that represents a stream of data. It produces successive items when you call its `next()` method on it. Once the items are exhausted, further calls to `next()` raise a `StopIteration` exception.

Under the hood

For the iterable, the key method that it must implement is `__iter__()`. This method is called to obtain an iterator from it.

For the iterator, the key method must implement i `__iter__()`. This method is called to obtain an iterator from it and `__next__()`.

`__iter__()`: Returns the iterator object itself. This is required to allow both iterators and iterables to be used in for loops and other iteration contexts.
`__next__()`: Returns the next item from the iterator. When there are no more items, it should raise `StopIteration`. When we are interacting wih an iterator wecan use either __next__()`: or next(itrator name).
So:
Creating an Iterator: An iterator is obtained from an iterable. When you iterate over an iterable (like a list), Python automatically calls the `__iter__()` method of the iterable to get an iterator (or iter()).

Iterating: The actual process of iteration over an iterable is done via an iterator. When you use a loop to go over an iterable, Python internally uses the iterator obtained from that iterable to fetch items one by one using __next__()

Example

```
my_list = [1, 2, 3]  # This is an iterable
my_iterator = iter(my_list)  # Obtaining an iterator from the iterable

for i in my_list:
    print(i)

for _ in my_iterator:
    print(__)

# Iterating using the iterator via next()
print(next(my_iterator))  # Outputs: 1
print(next(my_iterator))  # Outputs: 2
print(next(my_iterator))  # Outputs: 3
```

equivalently:

```
# Iterating using the iterator with the __next__() method
print(my_iterator.__next__())  # Outputs: 1
print(my_iterator.__next__())  # Outputs: 2
print(my_iterator.__next__())  # Outputs: 3
# The next call would raise StopIteration
```

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

Generators are created using a special type of function that uses the **yield** keyword to produce values on-the-fly. It also is "stateful", it keeps the state of the function between calls. It doesn't exit once it returns a value, it rather stays "alive" till all the expected values have been generated and returned to the caller.

Generator functions are like normal functions in most respects, and in fact are coded with normal def statements. When a **generator function** is called, it doesn't return a value, rather 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.

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
> >>>
```

What happens is that 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?

i.e. make this work:

```
def prime_dict(n):
    prime_generator = gen_primes()
    return {i: next(prime_generator) for i in range(1, n + 1)}
```

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 fib that will produce the next Fibonacci on demand.

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 **sensor_monitor(states)**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]

and when we call **sensor_monitor(runs)**

then the generator will output:

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

Answer

```
def sensor_monitor(states):
    gap = 0
    for state in states:
        if state == 1:
            yield gap
            gap = 0
        else:
            gap += 1
```

What is the issue?

The generators that we have encountered have **produced** values each time that they are called as well as preserving their state between those calls.

Here we have a situation where we need to **receive** inputs, possibly respond, and then wait for the next "signal" all the while preserving the sate of the function.

In the current problem we got all the inputs at once in the list states. But if we wanted to model a more realistic situation we would like our generator to receive inputs each time it is called.

Enter coroutines

Coroutines extend the concept of generators. While a generator is primarily used to produce values, coroutines can both produce and consume values. They are defined with `def` (like regular functions and generators), but use `yield`, `yield from`, or `await` to suspend their execution and transfer control back to the caller. The key difference is that coroutines can also receive data after yielding, which allows for two-way communication between the coroutine and its caller.

For the above problem imagine that we have a “producer” sending the ones and zeros to a “consumer” that is minding the gaps and sending out the gaps as before. Python has many powerful ways to do this including the async module and threading.

Let’s look at a simpler example here:

```
import random

def sensor_monitor():
    zero_count = 0
    while True:
        state = (yield zero_count) # state is where the generator get the value from the producer
        if state == 1:
            zero_count = 0
        else:
            zero_count += 1

def main():
    monitor = sensor_monitor() # monitor is the iterator
    next(monitor)  # Prime the generator

    for _ in range(20):  # Run for 20 iterations
        # Producer: Generate a random 0 or 1
        produced = random.randint(0, 1)
        print(f"Produced: {produced}")

        # Consumer: Process the produced value
        gap = monitor.send(produced)
        print(f"Consumed: {produced}, Gap: {gap}")

main()
```

and here is a step-by-step explanation:

1. **Initialization of the Sensor Monitor Generator**:

   monitor = sensor_monitor()

   next(monitor) # Prime the generator

   `sensor_monitor()` is called, which creates a generator object named `monitor`.

   `next(monitor)` is then called to 'prime' the generator. This step is necessary because the first call to a generator function doesn't execute any of the code in the function; it just returns a generator iterator. The `next()` function is used to advance the generator to the first `yield` statement, making it ready to receive values.

2. Starting the Loop:

```
for _ in range(20):  # Run for 20 iterations
```

A loop is set up to run for 20 iterations. This loop represents the main execution block where both production and consumption of values will take place.

3. Simulating the Producer:

```
produced = random.randint(0, 1)
print(f"Produced: {produced}")
```

Within the loop, a random integer (0 or 1) is generated, simulating the producer's action of generating sensor data.

This generated value is printed, indicating that it has been 'produced'.

4. Simulating the Consumer:

```
gap = monitor.send(produced)
print(f"Consumed: {produced}, Gap: {gap}")
```

The produced value is sent to the `monitor` generator using `monitor.send(produced)`. This is like the consumer receiving a value to process.

The generator (`monitor`) processes this value. It calculates the gap (number of zeros since the last one) and yields this gap back to the caller of `send`.

The gap value is then printed alongside the consumed value, indicating that the consumer has processed the value and determined the current gap.

5. Repeat:

The loop continues, repeating this process of producing a random value and consuming it by calculating the gap. Each iteration represents a new cycle of production and consumption.

6. End of Loop:

After 20 iterations, the loop ends. At this point, the program has simulated 20 cycles of producing and consuming sensor data.

This approach of interleaving the producer and consumer operations within a single loop is a straightforward way to demonstrate the concept without involving concurrency. It sequentially simulates the production of data and its consumption, showing how each piece of data is processed in order.

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**.

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__().