Python Generators Explained

Video

https://youtu.be/u3T7hmLthUU?si=usQ4E1XQTUArv2Fm

Notes

Generators vs Iterators

- An iterator is an object that allows you to loop through a sequence of data without having to store that sequence in memory
- A generator allows you to create your own iterator, which you can control the looping functionality of
  - New syntax (added in Python3)
  - Much easier to create than standard iterators
  - More “eloquent” iterator

Iterators Explained

- We use the `range()` function to create a loop because it does not store each value (0,1,2,3,…) in memory at once
  - `range()` returns an iterator
- Another example of an iterator is `map()`
  - `map()` is a function that applies another function to all the values in an iterable
  - It doesn’t actually store a new list that has all the results
  - It is an iterator that allows us to iterate through each result without actually storing them

- The values are generated as you iterate through the `map()` object that is returned when it is called

next()

- An iterator (such as a `for` loop) is actually calling `next()` on the iterator object that it has been given
- `next()` gives you the next value in an iterator
- Using `next()` is the same as calling `y.__next__()`

iter()

- If you try to call `next()` on a call of the `range()` function it will throw an exception as `range()` does not return an iterator
- To get the iterator from `range()` we have to use `iter()`
- `iter()` also has the dunder method equivalent `__iter__()`
- When you use a `for` loop, you are actually running `for i in next(iter(range(1,11))):`

Creating Legacy Iterators

- To create a iterator, you start by creating a class

```
class Iter:
    def __init__(self, num): # Constructor can contain anything you want
        self.max_num = num
```

- To make the class an iterator, you need to add two methods: `iter()` and `next()`

```
class Iter:
    def __init__(self, num):  # Constructor can contain anything you want
        self.max_num = num

    def __iter__(self):
        self.current = -1
        return self  # Return this instance of the object

    def __next__(self):
        self.current += 1
```

```
        if self.current >= self.max_num:
            raise StopIteration

        return self.current
```

- Next add the code to make use of this iterator

```
class Iter:
    def __init__(self, num):  # Constructor can contain anything you want
        self.max_num = num

    def __iter__(self):
        self.current = -1
        return self  # Return this instance of the object

    def __next__(self):
        self.current += 1

        if self.current >= self.max_num:
            raise StopIteration

        return self.current

# 1
x = Iter(5)

for i in x:
    print(i)


# 2
x = Iter(5)
itr = iter(x)  # Initialise the iterator by calling iter()
print(next(itr))
print(next(itr))
print(next(itr))
print(next(itr))
```

Creating Generators

- Rather than create a class, you create a function

```
import sys

def gen(n):  # Any parameters that you want
```

- Instead of using the `return` keyword, you use `yield`

```
import sys

def gen(n):  # Any parameters that you want
    # This is the exact same as the iterator class from the above section
    for i in range(n):
        yield n
```

- Then loop through it

```
import sys

def gen(n):  # Any parameters that you want
    # This is the exact same as the iterator class from the above section
    for i in range(n):
        yield n


for i in gen(5):
    print(i)
```

- When the `yield` keyword is hit, it pauses the execution of the function and returns the value to whatever is iterating through the generator object (in this case, the `for` loop)
    - When the generator pauses, the information about the function is saved in memory so it can be carried on
- The `for` loop then does whatever it needs to do with the returned value (in this case, `print()` it)
- The for loop then calls `next()` on the generator to get the next value
- The generator unpauses and yields the next value
- Essentially, the generator syntax makes the `next()` and `iter()` implemented for us so we do not need to manually implement them inside of a class
- You can also implement a generator with multiple `yield` statements rather than using a loop

```
import sys

def gen():  # Any parameters that you want
    yield 1
    yield 2
    yield 3
    yield 4


x = gen()
print(next(x))
print(next(x))
print(next(x))
print(next(x))
```

Generator Use Case

- You can loop through a sequence or some large amount of data without needing to store all of it
    - You would use a generator when you do not care about the data before or after something in an iteration, you only care about the current piece of data that you are dealing with
- If you were looking to see if a word exists in a file, you could read the whole file and check it but you do not need all of the file in memory at once
- You could use a generator to go line by line which will use up much less memory at any one time

Generator Comprehensions

- A way to create a generator without creating a function

```
x = (i for i in range(10))

print(x)
print(next(x))
print(next(x))
print(next(x))
print(next(x))
```

- In the code above, `x` will store a generator object