Theory: Custom generators

35 minutes 5 / 9 problems solved

Start practicing

1423 users solved this topic. Latest completion was about 9 hours ago.

§1. Generator functions

Imagine that in order to solve some problem, you need to obtain the first few multiples of some number `a` (for example, the first 4 multiples of 3 are 3, 6, 9, 12, etc.). The most straightforward way to do so is probably to define a function `multiples(a,n)` as follows:

```
def multiples(a, n):
    i = 1
    result = []
    while i <= n:
        result.append(a * i)
        i += 1
    return result


print(multiples(3, 5))
# Outputs [3, 6, 9, 12, 15]

print(multiples(2, 3))
# Outputs [2, 4, 6]
```

So, `multiples(a,n)` collects the first `n` multiples of `a` together in a list that is then returned. What are the disadvantages of such an approach?

Well, imagine that `n` is very large. If you get all the values at once, you will need to keep a very large list in memory. Is it necessary? It depends, but definitely not if you are going to use each value just once. Or maybe you don't even know exactly how many multiples you will eventually need, you just need to be able to get them one by one till some event happens.

For cases like this, **generator functions** are very helpful. A custom generator can be declared in the same way as a regular function with a single difference: the `return` keyword gets replaced with `yield`.

```
def multiples(a, n):
    i = 1
    while i <= n:
        yield a*i
        i += 1
```

When a regular function is called, Python goes back to its definition, runs the corresponding code with the provided argument values and returns the entire result with the `return` keyword to where the function is called from.

Generator functions, in turn, produce values one at a time, only when they are explicitly asked for a new one, rather than giving them all at once. Calling a generator doesn't immediately execute it. Instead, a generator object is returned that can be iterated over:

```
multiples(3, 10)
# <generator object multiples at 0x0000023501149048>
```

In order to get the generator function to actually compute its values, we need to explicitly ask for the next value by passing the generator into the `next()` function. Note that `yield` actually saves the state of the function, so that each time we ask the generator to produce a new value, execution continues from where it stopped, with the same variable values it had before yielding.

Current topic:

✓ Custom generators ···

Topic depends on:

✓ Declaring a function 10★ ··· Stage 1

✓ While loop 11★ ···

✓ List comprehension 5★ ···

Table of contents:

```python
# This is a generator.
multiples_of_three = multiples(3,5)

# It produces the first 5 multiples of 3 one by one:
print(next(multiples_of_three))
# 3
print(next(multiples_of_three))
# 6
print(next(multiples_of_three))
# 9
print(next(multiples_of_three))
# 12
print(next(multiples_of_three))
# 15
print(next(multiples_of_three))
# Error message: StopIteration
```

§2. Generator expressions

Another way of defining a generator is generator expressions, which are similar to list comprehensions. The only difference in the syntax are the brackets: one should use square brackets `[]` for list comprehension statements and the round ones `()` for defining a generator. Compare:

```python
numbers = [1, 2, 3]

my_generator = (n ** 2 for n in numbers)

print(next(my_generator))
# Outputs 1

print(next(my_generator))
# Outputs 4

print(next(my_generator))
# Outputs 9

# This is a list
my_list = [n ** 2 for n in numbers]

print(my_list)
# Outputs [1, 4, 9]
```

Generator expressions are very convenient to use in a `for` loop. A new value is automatically generated at each iteration:

```python
my_generator = (n ** 2 for n in numbers)

for n in my_generator:
    print(n)

# Outputs
# 1
# 4
# 9
```

§3. Why are generators useful?

So far, we've learned that generators produce a single value from a defined sequence only when they are explicitly asked to do so. This approach is called **lazy evaluation**.

Lazy evaluation makes the code much more memory efficient. Indeed, at each point in time, only values are produced and stored in memory one by one: the previous value is forgotten after we have moved to the next one and, therefore, doesn't take up space.

Keep in mind, however, that exactly because the previous value is forgotten when the new one needs to be generated, we can only go over the values once.

§4. Conclusions

Those were the basics of generators in Python. Let's sum it up:

- Generators allow one to declare a function that behaves like an iterator.
- Generators are lazy because they only give us a new value when we ask for it.
- There are two ways to create generators: generator functions and generator expressions.
- Generators are memory-efficient since they only require memory for the one value they yield.
- Generators can only be used once.