Generators

All Versions

Examples

Introduction

Generator expressions are similar to list, dictionary and set comprehensions, but are enclosed with parentheses. The parentheses do not have to be present when they are used as the sole argument for a function call.

```
expression = (x**2 for x in range(10))
```

This example generates the 10 first perfect squares, including 0 (in which x = 0).

Generator functions are similar to regular functions, except that they have one or more yield statements in their body. Such functions cannot return any values (however empty return s are allowed if you want to stop the generator early).

```
def function():
    for x in range(10):
        yield x**2
```

This generator function is equivalent to the previous generator expression, it outputs the same. **Note** : all generator expressions have their own *equivalent* functions, but not vice versa.

A generator expression can be used without parentheses if both parentheses would be repeated otherwise:

```
sum(i for i in range(10) if i % 2 == 0)   #Output: 20
any(x = 0 for x in foo)                   #Output: True or False depending on foo
type(a > b for a in foo if a % 2 == 1)    #Output: <class 'generator'>
```

Instead of:

```
sum((i for i in range(10) if i % 2 == 0))
any((x = 0 for x in foo))
type((a > b for a in foo if a % 2 == 1))
```

But not:

```
fooFunction((i for i in range(10) if i % 2 == 0,foo,bar)
return x = 0 for x in foo
barFunction(baz, a > b for a in foo if a % 2 == 1)
```

Calling a generator function produces a **generator object** , which can later be iterated over. Unlike other types of iterators, generator objects may only be traversed once.

```
g1 = function()
print(g1)  # Out: <generator object function at 0x1012e1888>
```

Notice that a generator's body is **not** immediately executed: when you call function() in the example above, it immediately returns a generator object, without executing even the first print statement. This allows generators to consume less memory than functions that return a list, and it allows creating generators that produce infinitely long sequences.

For this reason, generators are often used in data science, and other contexts involving large amounts of data. Another advantage is that other code can immediately use the values yielded by a generator, without waiting for the complete sequence to be produced.

However, if you need to use the values produced by a generator more than once, and if generating them costs more than storing, it may be better to store the yielded values as a list than to re-generate the sequence.

Typically a generator object is used in a loop, or in any function that requires an iterable:

```
for x in g1:
    print("Received", x)

# Output:
# Received 0
# Received 1
# Received 4
# Received 9
# Received 16
# Received 25
# Received 36
# Received 49
# Received 64
# Received 81

arr1 = list(g1)
# arr1 = [], because the loop above already consumed all the values.
g2 = function()
arr2 = list(g2)  # arr2 = [0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

Since generator objects are iterators, one can iterate over them manually using the next() function. It will return the yielded values one by one on each subsequent invocation.

Under the hood, each time you call next() on a generator, Python executes statements in the body of the generator function until it hits the next yield statement. At this point it returns the argument of the yield command, and remembers the point where that happened. Calling next() once again will resume execution from that point and continue until the next yield statement.

If Python reaches the end of the generator function without encountering any more yield s, a StopIteration exception is raised (this is normal, all iterators behave in the same way).

```
g3 = function()
a = next(g3)  # x becomes 0
b = next(g3)  # y becomes 1
c = next(g3)  # z becomes 2
...
j = next(g3)  # Raises StopIteration, z remains undefined
```

Note that in Python 2 generator objects had .next() methods that could be used to iterate through the yielded values manually. In Python 3 this method was replaced with the .__next__() standard for all iterators.

Infinite sequences

Generators can be used to represent infinite sequences:

```
def integers_starting_from(n):
    while True:
        yield n
        n += 1

natural_numbers = integers_starting_from(1)
```

Infinite sequence of numbers as above can also be generated with the help of itertools.count . The above code could be written as below

```
natural_numbers = itertools.count(1)
```

You can use generator comprehensions on infinite generators to produce new generators:

```
multiples_of_two = (x * 2 for x in natural_numbers)
multiples_of_three = (x for x in natural_numbers if x % 3 == 0)
```

Be aware that an infinite generator does not have an end, so passing it to any function that will attempt to consume the generator entirely will have **dire consequences** :

```
list(multiples_of_two)  # will never terminate, or raise an OS-specific error
```

Instead, use list/set comprehensions with range (or xrange for python < 3.0):

```
first_five_multiples_of_three = [next(multiples_of_three) for _ in range(5)]
# [3, 6, 9, 12, 15]
```

or use itertools.islice() to slice the iterator to a subset:

```
from itertools import islice
multiples_of_four = (x * 4 for x in integers_starting_from(1))
first_five_multiples_of_four = list(islice(multiples_of_four, 5))
# [4, 8, 12, 16, 20]
```

Note that the original generator is updated too, just like all other generators coming from the same "root":

```
next(natural_numbers)    # yields 16
next(multiples_of_two)   # yields 34
next(multiples_of_four)  # yields 24
```

An infinite sequence can also be iterated with a for -loop . Make sure to include a conditional break statement so that the loop would terminate eventually:

```
for idx, number in enumerate(multiplies_of_two):
    print(number)
    if idx == 9:
        break  # stop after taking the first 10 multiplies of two
```

Classic example - Fibonacci numbers

```
import itertools

def fibonacci():
    a, b = 1, 1
    while True:
        yield a
        a, b = b, a + b

first_ten_fibs = list(itertools.islice(fibonacci(), 10))
# [1, 1, 2, 3, 5, 8, 13, 21, 34, 55]

def nth_fib(n):
    return next(itertools.islice(fibonacci(), n - 1, n))

ninety_nineth_fib = nth_fib(99)  # 354224848179261915075
```

Sending objects to a generator

In addition to receiving values from a generator, it is possible to *send* an object to a generator using the send() method.

```
def accumulator():
    total = 0
    value = None
    while True:
        # receive sent value
        value = yield total
        if value is None: break
        # aggregate values
        total += value

generator = accumulator()

# advance until the first "yield"
next(generator)      # 0

# from this point on, the generator aggregates values
generator.send(1)    # 1
generator.send(10)   # 11
generator.send(100)  # 111
# ...

# Calling next(generator) is equivalent to calling generator.send(None)
next(generator)      # StopIteration
```

What happens here is the following:

- When you first call next(generator) , the program advances to the first yield statement, and returns the value of total at that point, which is 0. The execution of the generator suspends at this point.
- When you then call generator.send(x) , the interpreter takes the argument x and makes it the return value of the last yield statement, which gets assigned to value . The generator then proceeds as usual, until it yields the next value.
- When you finally call next(generator) , the program treats this as if you're sending None to the generator. There is nothing special about None , however, this example uses None as a special value to ask the generator to stop.

Yielding all values from another iterable

Python 3.x ≥ 3.3

Use yield from if you want to yield all values from another iterable:

```
def foob(x):
    yield from range(x * 2)
    yield from range(2)

list(foob(5))  # [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 0, 1]
```

This works with generators as well.

```
def fibto(n):
    a, b = 1, 1
    while True:
        if a >= n: break
        yield a
        a, b = b, a + b

def usefib():
    yield from fibto(10)
    yield from fibto(20)

list(usefib())  # [1, 1, 2, 3, 5, 8, 1, 1, 2, 3, 5, 8, 13]
```

Iteration

A generator object supports the *iterator protocol* . That is, it provides a next() method (__next__() in Python 3.x), which is used to step through its execution, and its __iter__ method returns itself. This means that a generator can be used in any language construct which supports generic iterable objects.

```
# naive partial implementation of the Python 2.x xrange()
def xrange(n):
    i = 0
    while i < n:
        yield i
        i += 1

# looping
for i in xrange(10):
    print(i)  # prints the values 0, 1, ..., 9

# unpacking
a, b, c = xrange(3)  # 0, 1, 2

# building a list
l = list(xrange(10))  # [0, 1, ..., 9]
```

Coroutines

Generators can be used to implement coroutines:

```
# create and advance generator to the first yield
def coroutine(func):
    def start(*args,**kwargs):
        cr = func(*args,**kwargs)
        next(cr)
        return cr
    return start

# example coroutine
@coroutine
def adder(sum = 0):
    while True:
        x = yield sum
        sum += x

# example use
s = adder()
s.send(1) # 1
s.send(2) # 3
```

Coroutines are commonly used to implement state machines, as they are primarily useful for creating single-method procedures that require a state to function properly. They operate on an existing state and return the value obtained on completion of the operation.

The next() function

The next() built-in is a convenient wrapper which can be used to receive a value from any iterator (including a generator iterator) and to provide a default value in case the iterator is exhausted.

```
def nums():
    yield 1
    yield 2
    yield 3
generator = nums()

next(generator, None)  # 1
next(generator, None)  # 2
next(generator, None)  # 3
next(generator, None)  # None
next(generator, None)  # None
# ...
```

The syntax is next(iterator[, default]) . If iterator ends and a default value was passed, it is returned. If no default was provided, StopIteration is raised.

Yield with recursion: recursively listing all files in a directory

First, import the libraries that work with files:

```
from os import listdir
from os.path import isfile, join, exists
```

A helper function to read only files from a directory:

```
def get_files(path):
    for file in listdir(path):
        full_path = join(path, file)
        if isfile(full_path):
            if exists(full_path):
                yield full_path
```

Another helper function to get only the subdirectories:

```
def get_directories(path):
    for directory in listdir(path):
        full_path = join(path, directory)
        if not isfile(full_path):
            if exists(full_path):
                yield full_path
```

Now use these functions to recursively get all files within a directory and all its subdirectories (using generators):

```
def get_files_recursive(directory):
    for file in get_files(directory):
        yield file
    for subdirectory in get_directories(directory):
        for file in get_files_recursive(subdirectory): # here the recursive call
            yield file
```

This function can be simplified using yield from :

```
def get_files_recursive(directory):
    yield from get_files(directory)
    for subdirectory in get_directories(directory):
        yield from get_files_recursive(subdirectory)
```

Generator expressions

It's possible to create generator iterators using a comprehension-like syntax.

```
generator = (i * 2 for i in range(3))

next(generator)  # 0
next(generator)  # 2
next(generator)  # 4
next(generator)  # raises StopIteration
```

If a function doesn't necessarily need to be passed a list, you can save on characters (and improve readability) by placing a generator expression inside a function call. The parenthesis from the function call implicitly make your expression a generator expression.

```
sum(i ** 2 for i in range(4))  # 0^2 + 1^2 + 2^2 + 3^2 = 0 + 1 + 4 + 9 = 14
```

Additionally, you will save on memory because instead of loading the entire list you are iterating over ([0, 1, 2, 3] in the above example), the generator allows Python to use values as needed.

Refactoring list-building code

Suppose you have complex code that creates and returns a list by starting with a blank list and repeatedly appending to it:

```
def create():
    result = []
    # logic here...
    result.append(value) # possibly in several places
    # more logic...
    return result # possibly in several places

values = create()
```

When it's not practical to replace the inner logic with a list comprehension, you can turn the entire function into a generator in-place, and then collect the results:

```
def create_gen():
    # logic...
    yield value
    # more logic
    return # not needed if at the end of the function, of course

values = list(create_gen())
```

If the logic is recursive, use `yield from` to include all the values from the recursive call in a "flattened" result:

```
def preorder_traversal(node):
    yield node.value
    for child in node.children:
        yield from preorder_traversal(child)
```

Searching

The `next` function is useful even without iterating. Passing a generator expression to `next` is a quick way to search for the first occurrence of an element matching some predicate. Procedural code like

```
def find_and_transform(sequence, predicate, func):
    for element in sequence:
        if predicate(element):
            return func(element)
    raise ValueError

item = find_and_transform(my_sequence, my_predicate, my_func)
```

can be replaced with:

```
item = next(my_func(x) for x in my_sequence if my_predicate(x))
# StopIteration will be raised if there are no matches; this exception can
# be caught and transformed, if desired.
```

For this purpose, it may be desirable to create an alias, such as `first = next` , or a wrapper function to convert the exception:

```
def first(generator):
    try:
        return next(generator)
    except StopIteration:
        raise ValueError
```

Using a generator to find Fibonacci Numbers

A practical use case of a generator is to iterate through values of an infinite series. Here's an example of finding the first ten terms of the Fibonacci Sequence .

```
def fib(a=0, b=1):
    """Generator that yields Fibonacci numbers. `a` and `b` are the seed values"""
    while True:
        yield a
        a, b = b, a + b

f = fib()
print(', '.join(str(next(f)) for _ in range(10)))
```

```
0, 1, 1, 2, 3, 5, 8, 13, 21, 34
```

Iterating over generators in parallel

To iterate over several generators in parallel, use the zip builtin:

```
for x, y in zip(a,b):
    print(x,y)
```

Results in:

```
1 x
2 y
3 z
```

In python 2 you should use itertools.izip instead. Here we can also see that the all the zip functions yield tuples.

Note that zip will stop iterating as soon as one of the iterables runs out of items. If you'd like to iterate for as long as the longest iterable, use itertools.zip_longest() .

Syntax

```
yield <expr>
```

```
yield from <expr>
```

```
<var> = yield <expr>
```

```
next( <iter> )
```

Parameters

Remarks