Project Report

Python Generators

Debabrata Bhattacharya

Contents

Python Generators....3

What are they? 3

Yield statement 4

Benefits 4

Comprehensions....5

Advanced Generator functions 6

Code 7

Output 14

Test Results....16

Conclusion 17

Python Generators

What are they?

Introduced with PEP 255, generator functions are a special kind of function that return a lazy iterator. These are objects that you can loop over like a list. However, unlike lists, lazy iterators do not store their contents in memory.

Generator functions look and act just like regular functions, but with one defining characteristic. Generator functions use the Python yield keyword instead of return.

This looks like a typical function definition, except for the Python yield statement and the code that follows it. yield indicates where a value is sent back to the caller, but unlike return, you don't exit the function afterward.

```
def infinite_sequence():
    num = 0
    while True:
        yield num
        num += 1
```

Instead, the state of the function is remembered. That way, when next() is called on a generator object (either explicitly or implicitly within a for loop), the previously yielded variable num is incremented, and then yielded again. Since generator functions look like other functions and act very similarly to them, you can assume that generator expressions are very similar to other comprehensions available in Python.

Generators, like all iterators, can be exhausted. Unless your generator is infinite, you can iterate through it one time only. Once all values have been evaluated, iteration will stop and the for loop will exit. If you used next(), then instead you'll get an explicit StopIteration exception.

Yield statement

On the whole, yield is a fairly simple statement. Its primary job is to control the flow of a generator function in a way that's similar to return statements. As briefly mentioned above, though, the Python yield statement has a few tricks up its sleeve.

When you call a generator function or use a generator expression, you return a special iterator called a generator. You can assign this generator to a variable in order to use it. When you call special methods on the generator, such as next(), the code within the function is executed up to yield.

When the Python yield statement is hit, the program suspends function execution and returns the yielded value to the caller. (In contrast, return stops function execution completely.) When a function is suspended, the state of that function is saved. This includes any variable bindings local to the generator, the instruction pointer, the internal stack, and any exception handling.

This allows you to resume function execution whenever you call one of the generator's methods. In this way, all function evaluation picks back up right after yield.

Benefits

1. Reading Large Files
2. Generating an Infinite Sequence

Comprehensions

You can also define a generator expression (also called a generator comprehension), which has a very similar syntax to list comprehensions. In this way, you can use the generator without calling a function:

```
csv_gen = (row for row in open(file_name))
```

Like list comprehensions, generator expressions allow you to quickly create a generator object in just a few lines of code. They're also useful in the same cases where list comprehensions are used, with an added benefit: you can create them without building and holding the entire object in memory before iteration. In other words, you'll have no memory penalty when you use generator expressions.

Advanced Generator functions

- .send()
  - Send data back to the generator
- .throw()
  - Throw an exception
- .close()
  - Terminate the generator

Example:

```
def infinite_palindromes():
    num = 0
    while True:
        if is_palindrome(num):
            i = (yield num)
            if i is not None:
                num = i
        num = num + 1
  def use_send():
    pal_gen = infinite_palindromes()
    for i in pal_gen:
        pals.append(i)
        if i == 10000100001:
            break
        digits = len(str(i))
        pal_gen.send(10**(digits))
```

Code

```
import csv
import logging
import logging.config
from json import load as jload

""" Configure logger lg with config for appLogger from config.json["logging"]
"""
with open('config.json', 'r') as f:
        config = jload(f)
        logging.config.dictConfig(config["logging"])
lg = logging.getLogger('appLogger')
# lg.debug("This is a debug message")

class Generators(object):
    def using_generators(self):
        def reading_large_files():
            """ Introduced with PEP 255, generator functions are a special kin
d of function that return a lazy iterator. These are objects that you can loop
 over like a list. However, unlike lists, lazy iterators do not store their co
ntents in memory.  """
            def reading_large_files1():
                """ A common use case of generators is to work with data strea
ms or large files, like CSV files. These text files separate data into columns
 by using commas. This format is a common way to share data. Now, what if you
want to count the number of rows in a CSV file?  """
                with open("techcrunch.csv", "r") as f:
                    csv_gen = csv.reader(f)
                    row_count = 0
                    for row in csv_gen:# pylint: disable=unused-variable
                        row_count += 1
                    row_count_string = f"Row count is {row_count} in reading_l
arge_files1"
                    lg.info(row_count_string)
                    return row_count

            def reading_large_files2():
                """ What’s happening here? Well, you’ve essentially turned csv
_reader() into a generator function. This version opens a file, loops through
each line, and yields each row, instead of returning it. """
                def csv_reader(filename):
                    with open(filename, "r") as f:
                        for row in f:
                            yield row
                csv_gen = csv_reader("techcrunch.csv")
                row_count = 0
                for row in csv_gen:# pylint: disable=unused-variable
                    row_count += 1
```

```
                row_count_string = f"Row count is {row_count} in reading_large
_files2"
                lg.info(row_count_string)
                return row_count

            def reading_large_files3():
                """ You can also define a generator expression (also called a
generator comprehension), which has a very similar syntax to list comprehensio
ns. In this way, you can use the generator without calling a function: """
                csv_gen = (row for row in open("techcrunch.csv", "r"))
                row_count = 0
                for row in csv_gen:# pylint: disable=unused-variable
                    row_count += 1
                row_count_string = f"Row count is {row_count} in reading_large
_files3"
                lg.info(row_count_string)
                return row_count

            return reading_large_files1(),reading_large_files2(),reading_large
_files3()

        def generating_an_infinite_sequence():
            """ First, you initialize the variable num and start an infinite l
oop. Then, you immediately yield num so that you can capture the initial state
. This mimics the action of range().After yield, you increment num by 1. If yo
u try this with a for loop, then you’ll see that it really does seem infinite
"""
            def infinite_sequence():
                num = 0
                while True:
                    yield num
                    num += 1
            nums = "Infinite sequence nums = "
            for i in infinite_sequence():
                nums += str(i) + "   "
                if i == 999: break
            lg.info(nums)
            return nums

        def generating_an_infinite_sequence2():
            """ Here, you have a generator called gen, which you manually iter
ate over by repeatedly calling next(). This works as a great sanity check to m
ake sure your generators are producing the output you expect. """
            def infinite_sequence():
                num = 0
                while True:
                    yield num
                    num+=1
```

```
            nums = "Second infinite sequence = "
            gen = infinite_sequence()
            while True:
                i = next(gen)
                nums += str(i) + "  "
                if i == 999: break
            lg.info(nums)
            return nums

        def detecting_palindromes():
            def infinite_sequence():
                num = 0
                while True:
                    yield num
                    num+=1
            def is_palindrome(num):
                # skip single digit inputs
                if num // 10 == 0:
                    return False
                temp = num
                reversed_num = 0
                # reverse the input num
                while temp!=0:
                    reversed_num = (reversed_num*10) + (temp % 10)
                    temp = temp // 10
                # check if reversed num and num are the same number
                if num == reversed_num:
                    return True
                else:
                    return False
            pal_nums = "Palindrome number sequence: "
            for i in infinite_sequence():
                if i == 102202:
                    break
                pal = is_palindrome(i)
                if pal:
                    pal_nums += str(i) + "  "
            lg.info(pal_nums)
            return pal_nums
        return [reading_large_files(),generating_an_infinite_sequence(),genera
ting_an_infinite_sequence2(),detecting_palindromes()]
    def understanding_generators(self):
        def building_generators_with_generator_expressions():
            nums_squared_lc = [i**2 for i in range(5)]
            nums_squared_gc = (i**2 for i in range(5))
            lg.info('type of nums_squared_lc:{}'.format(type(nums_squared_lc))
)
```

```
            lg.info('type of nums_squared_gc:{}'.format(type(nums_squared_gc))
)
            return [str(type(nums_squared_lc)), str(type(nums_squared_gc))]
        types = building_generators_with_generator_expressions()
        def profiling_generator_performance():
            from sys import getsizeof
            nums_squared_lc = [i**2 for i in range(10000)]
            nums_squared_gc = (i**2 for i in range(10000))
            # Add string info
            lg.info('size of lc:{}'.format(getsizeof(nums_squared_lc)))
            lg.info('size of gc:{}'.format(getsizeof(nums_squared_gc)))
            from cProfile import run
            run('sum([i**2 for i in range(10000)])','lc.profile')
            import pstats
            p = pstats.Stats('lc.profile')
            # print number of calls
            lg.info(('Number of function calls for lc:'+' {}'.format(p.prim_ca
lls)))
            run('sum((i**2 for i in range(10000)))','gc.profile')
            q = pstats.Stats('gc.profile')
            lg.info(('Number of function calls for gc:'+' {}'.format(q.prim_ca
lls)))
            return [getsizeof(nums_squared_lc),getsizeof(nums_squared_gc),p.pr
im_calls,q.prim_calls]
        stats = profiling_generator_performance()
        return [types, stats]
    def understanding_yeild_statement(self):
        """ On the whole, yield is a fairly simple statement. Its primary job
is to control the flow of a generator function in a way that's similar to retu
rn statements. As briefly mentioned above, though, the Python yield statement
has a few tricks up its sleeve.When you call a generator function or use a gen
erator expression, you return a special iterator called a generator. You can a
ssign this generator to a variable in order to use it. When you call special m
ethods on the generator, such as next(), the code within the function is execu
ted up to yield.When the Python yield statement is hit, the program suspends f
unction execution and returns the yielded value to the caller. (In contrast, r
eturn stops function execution completely.) When a function is suspended, the
state of that function is saved. This includes any variable bindings local to
the generator, the instruction pointer, the internal stack, and any exception
handling.This allows you to resume function execution whenever you call one of
 the generator's methods. In this way, all function evaluation picks back up r
ight after yield. """
        def multi_yield():
            yield_str = "This is the first string"
            yield yield_str
            yield_str = "This is the second string"
            yield yield_str
        multi_obj = multi_yield()
```

```python
        strings = []
        for i in range(10): # pylint: disable=unused-variable
            try:
                strings.append(next(multi_obj))
            except StopIteration:
                lg.error("Stop Iteration error: generator exhausted")
                break
        lg.info(strings)
        return strings
    def adv_generator_methods(self):
        """ Using send(), throw(), and close() """
        pals = []
        pals2 = []
        pals3 = []
        def is_palindrome(num):
            if num // 10 == 0:
                return False
            temp = num
            rev = 0
            while temp!=0:
                rev = (rev *10) + (temp % 10)
                temp = temp // 10
            if rev == num:
                return True
            else:
                return False
        def infinite_palindromes():
            num = 0
            while True:
                if is_palindrome(num):
                    i = (yield num)
                    if i is not None:
                        num = i
                num = num + 1
        def use_send():
            pal_gen = infinite_palindromes()
            for i in pal_gen:
                pals.append(i)
                if i == 10000100001:
                    break
                digits = len(str(i))
                pal_gen.send(10**(digits))
        def use_throw():
            pal_gen = infinite_palindromes()
            for i in pal_gen:
                pals2.append(i)
                digits = len(str(i))
                if digits == 5:
```

```
                    pal_gen.throw(ValueError("we don't like this large palindr
ome"))
                pal_gen.send(10**(digits))
        def use_close():
            pal_gen = infinite_palindromes()
            for i in pal_gen:
                pals3.append(i)
                digits = len(str(i))
                if digits == 5:
                    pal_gen.close()
                pal_gen.send(10**(digits))
        use_send()
        lg.info(pals)
        try:
            use_throw()
        except ValueError:
            lg.error("ValueError exception thrown by generator")
        lg.info(pals2)
        try:
            use_close()
        except StopIteration:
            lg.error("StopIteration exception thrown by generator")
        lg.info(pals3)
        return pals, pals2, pals3
    def data_pipelines(self):
        filename = "techcrunch.csv"
        # read every line in file
        lines = (line for line in open(filename))
        # split each line into a list of values
        list_line = (s.rstrip().split(",") for s in lines )
        # extract the column names
        cols = next(list_line)
        # create a dict of values from lists
        company_dicts = (dict(zip(cols,data)) for data in list_line)
        # Filter out irrelevant data
        funding = (
            int(company_dict["raisedAmt"])
            for company_dict in company_dicts
            if company_dict["round"]  == "a"
        )
        # calculate total and avg
        total_amt_raised = sum(funding)
        result_sum = f"Total series A fundraising : ${total_amt_raised}"
        lg.info(result_sum)
        # avg raised per company
        """ Find out number of companies
        \nDivide total_amt_raised by number of companies"""
        def dict_gen():
```

```python
            # read every line in file
            lines = (line for line in open(filename))
            # split each line into a list of values
            list_line = (s.rstrip().split(",") for s in lines )
            # extract the column names
            cols = next(list_line)
            # create a dict of values from lists
            company_dicts = (dict(zip(cols,data)) for data in list_line)
            return company_dicts
        company_dicts = dict_gen()

        num_comps = len(set((
            str(company_dict["company"])
            for company_dict in company_dicts
            if company_dict["round"]  == "a"
        )))
        avg = total_amt_raised//num_comps
        result_avg = f"Average amount raised by company = {avg}"
        lg.info("Number of companies:"+str(num_comps))
        lg.info(result_avg)
        return total_amt_raised, num_comps, avg



gen = Generators()
gen.using_generators()
gen.understanding_generators()
gen.understanding_yeild_statement()
gen.adv_generator_methods()
gen.data_pipelines()
```

Output

```
appLogger - 2020-03-15 14:09:41,305-8356-INFO-
Row count is 1461 in reading_large_files1
appLogger - 2020-03-15 14:09:41,307-8356-INFO-
Row count is 1461 in reading_large_files2
appLogger - 2020-03-15 14:09:41,308-8356-INFO-
Row count is 1461 in reading_large_files3
appLogger - 2020-03-15 14:09:41,309-8356-INFO-
Infinite sequence nums = 0  1  2  3  4  5  6  7  8  9  10  11  12  13  14  15
 16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  3
5  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54
 55  56  57  58  59  60  61  62  63  64  65  66  67  68  69  70  71  72  73  7
4  75  76  77  78  79  80  81  82  83  84  85  86  87  88  89  90  91  92  93
 94  95  96  97  98  99  100  101  102  103  …  998  999
appLogger - 2020-03-15 14:09:41,309-8356-INFO-
Second infinite sequence = 0  1  2  3  4  5  6  7  8  9  10  11  12  13  14  1
5  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34
 35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  5
4  55  56  57  58  59  60  61  62  63  64  65  66  67  68  69  70  71  72  73
 74  75  76  77  78  79  80  81  82  83  84  85  86  87  88  89  90  91  92  9
3  94  95  96  97  98  99  100  101  102  103  … 998  999
appLogger - 2020-03-15 14:09:41,721-8356-INFO-
Palindrome number sequence: 11  22  33  44  55  66  77  88  99  101  111  121
 131  141  151  161  171  181  191  202  212  222  232  242  252  262  272  28
2  292  303  313  323  333  343  353  363  373  383  393  404  414  424  434
444  454  464  474  484  494  505  515  525  535  545  555  565  575  585  595
  606  616  626  636  646  656  666  676  686  696  …
99799  99899  99999  100001  101101  102201
appLogger - 2020-03-15 14:09:41,722-8356-INFO-
type of nums_squared_lc:<class 'list'>
appLogger - 2020-03-15 14:09:41,722-8356-INFO-
type of nums_squared_gc:<class 'generator'>
appLogger - 2020-03-15 14:09:41,796-8356-INFO-size of lc:87624
appLogger - 2020-03-15 14:09:41,797-8356-INFO-size of gc:120
appLogger - 2020-03-15 14:09:41,881-8356-INFO-
Number of function calls for lc: 5
appLogger - 2020-03-15 14:09:41,894-8356-INFO-
Number of function calls for gc: 10005
appLogger - 2020-03-15 14:09:41,894-8356-ERROR-
Stop Iteration error: generator exhausted
appLogger - 2020-03-15 14:09:41,894-8356-INFO-
['This is the first string', 'This is the second string']
appLogger - 2020-03-15 14:09:43,581-8356-INFO-
[11, 111, 1111, 10101, 101101, 1001001, 10011001, 100010001, 1000110001, 10000
100001]
appLogger - 2020-03-15 14:09:43,582-8356-ERROR-
ValueError exception thrown by generator
appLogger - 2020-03-15 14:09:43,582-8356-INFO-[11, 111, 1111, 10101]
```

```
appLogger - 2020-03-15 14:09:43,583-8356-ERROR-
StopIteration exception thrown by generator
appLogger - 2020-03-15 14:09:43,583-8356-INFO-[11, 111, 1111, 10101]
appLogger - 2020-03-15 14:09:43,614-8356-INFO-
Total series A fundraising : $4376015000
appLogger - 2020-03-15 14:09:43,620-8356-INFO-Number of companies:563
appLogger - 2020-03-15 14:09:43,620-8356-INFO-
Average amount raised by company = 7772673
```

Test Results

```
(base) J:\Education\Code\Python\Python-Generators>pytest test_gen_script.py -vv
================================================= test session starts =================================================
platform win32 -- Python 3.7.3, pytest-5.0.1, py-1.8.0, pluggy-0.12.0 -- C:\ProgramData\Anaconda3\python.exe
cachedir: .pytest_cache
rootdir: J:\Education\Code\Python\Python-Generators
plugins: arraydiff-0.3, doctestplus-0.3.0, openfiles-0.3.2, remotedata-0.3.1
collected 5 items

test_gen_script.py::TestObject::test_using_generators PASSED                                                    [ 20%]
test_gen_script.py::TestObject::test_understanding_generators PASSED                                            [ 40%]
test_gen_script.py::TestObject::test_understanding_yeild_statement PASSED                                       [ 60%]
test_gen_script.py::TestObject::test_adv_generator_methods PASSED                                               [ 80%]
test_gen_script.py::TestObject::test_data_pipelines PASSED                                                      [100%]

============================================== 5 passed in 1.97 seconds ===============================================
```

Conclusion

Python generator functions serve as an important tool in the implementation of algorithms where memory usage is a concern, asynchronous evaluation matters, and functional overhead doesn't matter.