Chapter 04 - Comprehensions and Generators

May 8, 2021

0.1 Overview

- python provides special syntax, called `comprehensions` for iterating through these types and creating derivative data structures
- this style of processing is extended to functions with `generators` which enable a stream of values to be incrementally returned by a function
- the result of a call to a generator function can be used where an iterator is apporpriate (loops, starred expressions)

0.2 Item 27: Use Comprehensions Instead of `map` and `filter`

- you should perfer list comprehensions over `map` built in function because `map` requires lambda
- so unless you have only one argument, dont use `lambda` stuff
- with comprehensions you can also filter things

[2]:
```
a = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
even_squares = [x**2 for x in a if x % 2 == 0]
print(even_squares)
```

```
[4, 16, 36, 64, 100]
```

- dictionary comprehensions also exist

[3]:
```
even_squares_dict = {x:x**2 for x in a if x % 2 == 0}
print(even_squares_dict)
```

```
{2: 4, 4: 16, 6: 36, 8: 64, 10: 100}
```

0.3 Avoid More Than Two Control Subexpressions in Comprehensions

- it is fine to use two o `for` loops but try not to use any more

[4]:
```
matrix = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
flat = [x for row in matrix for x in row]
print(flat)
```

```
[1, 2, 3, 4, 5, 6, 7, 8, 9]
```

- another reasonable usage of multiple loops involves replicating the two-level-deep layout of the input `list`
- for example, if you wanted the square value in each cell of a two-dimensional matrix

[5]:
```python
squared = [[x**2 for x in row] for row in matrix]
print(squared)
```

```
[[1, 4, 9], [16, 25, 36], [49, 64, 81]]
```

- be careful of multi-line comprehensions, espically if they start looking like normal loops

[8]:
```python
def bad_comprehension():
    flat = [x for sublist1 in my_lists
        for sublist2 in sublist1
        for x in sublist2]

    flat = []
    for sublist1 in my_lists:
        for sublist2 in sublist1:
            flat.extend(sublist2)
```

0.4 Item 29: Avoid Repeated Work in Comprehensions by Using Assignment Expressions

- Amazing you can combine walrus operator along with a list comprehension
- the assignment expression `batches := get_batches(...)` allows us to look up the value for each order key in the `stock` dictionary a single time
- `get_batches` then store its corresponding value in the `batches` varible
- we can then refrence that variable elsewhere in the comprehension to construct the dict's contents instead of having to call `get_batches` a second time

[12]:
```python
def avoiding_repeated_work():

    result = {}
    for name in order:
        count = stock.get(name, 0)
        batches = get_batches(count, 8)

    has_bug = {name: get_batches(stock.get(name, 0), 4)
               for name in order
               if get_batches(stock.get(name, 0), 8)}

    # we can further reduce the ugliness with walrus operator
    found = {name: batches for name in order
             if (batches := get_batches(stock.get(name, 0), 8))}

    result = {name: tenth for name, count in stock.items()
             if (tenth := count // 10) > 0}
```

- if a comprehension uses the walrus operator in the value part of the comprehension and does not have a condition, it will leak the loop variable into the containing scope
- the leakage of the loop variable is simmilar to what happens with a normal `for` loop

[14]:
```python
stock = {
 'nails': 125,
 'screws': 35,
 'wingnuts': 8,
 'washers': 24,
}

for count in stock.values(): # Leeks loop variable
    pass
print(f'Last item of {list(stock.values())} is {count}')
```

```
Last item of [125, 35, 8, 24] is 24
```

- similar leakage does not happen for the loop variables from comprehensions

[16]:
```python
half = [count // 2 for count in stock.values()]
print(half) # Works
print(count) # Exception because loop variable dident leak
```

```
[62, 17, 4, 12]
24
```

- its better not to leak loop variables, so I recommend using assignment expressions only in the condition part of a comprehension
- assignment expression also work the same way in generator expressions
- the difference is using the `()` instead of `[]`

[20]:
```python
order = ['screws', 'wingnuts', 'clips']
def get_batches(count, size):
    return count // size

found = ((name, batches) for name in order
          if (batches := get_batches(stock.get(name, 0), 8)))

print(next(found))
print(next(found))
```

```
('screws', 4)
('wingnuts', 1)
```

- Although it's possible to use an assignment expression outside of a comprehension or generator expression's condition, you should avoid doing so

0.5 Item 30: Consider Generators Instead of Returning Lists

[21]:
```python
def index_words(text):
    result = []
    if text:
        result.append(0)
    for index, letter in enumerate(text):
```

```
        if letter == ' ':
            result.append(index + 1)
    return result
```

- normal returns such as the list show above has two problems with it
- the **first** proble is that the code is dense and noise
- look at the times `append` is being called
- the `result.append` deemphasizes the value being added to the `list (index + 1)`
- there is one line for creating the result `list` and another for returning it

- the function `index_words` can be rewritten as shown below

[22]:
```
def index_words_iter(text):
    if text:
        yield 0
    for index, letter in enumerate(text):
        if letter == ' ':
            yield index + 1
```

- when called, a generator function does not actually run but instead immediately returns an iterator
- with each call to the `next` built-in function, the iterator advances the generator to its next `yeild` expression
- each value passed to `yield` by the generator is returned by the iterator to the caller

[24]:
```
address = 'Four score and seven years ago...'

it = index_words_iter(address)
print(next(it))
print(next(it))
```

```
0
5
```

- the `index_words_iter` function is significantly easier to read because all itneractions with the result `list` have been eliminated
- results are passed to `yield` expression instead

- you can easily conver a generator to a list

[25]:
```
result = list(index_words_iter(address))
print(result[:10])
```

```
[0, 5, 11, 15, 21, 27]
```

- the **second** proble with the `index_words` is that it requires all results to be stored in the `list` before being returned
- in contrast, a generator version of this function can easily be adapted to take inputs of arbitrary lenght due to its bounded memory requirements

[26]:
```
def index_file(handle):
    offset = 0
    for line in handle:
        if line:
            yield offset
        for letter in line:
            offset += 1
            if letter == ' ':
                yield offset
```

- the working memeory for this function is limited to the maximum lenght of one line of input

[28]:
```
def using_generator_index_file():
    with open('address.txt', 'r') as f:
        it = index_file(f)
        results = itertools.islice(it, 0, 10)
        print(list(results))
```

- the gotcha with defining generators like this is that the callers must be aware that the iterators returned are stateful and cant be reused

0.6 Item 31: Be Denfensive When Iterating Over Arguments

- imagine you has a function that told you what percentage of the overall turism a city recieves

[29]:
```
def normalize(numbers):
    total = sum(numbers)
    result = []
    for value in numbers:
        percent = 100 * value / total
        result.append(percent)
    return result

visits = [15, 35, 80]
percentages = normalize(visits)
print(percentages)
assert sum(percentages) == 100.0
```

```
[11.538461538461538, 26.923076923076923, 61.53846153846154]
```

- to scale this up, we need to be able to read from a text file

[31]:
```
def read_visits(data_path):
    with open(data_path) as f:
        for line in f:
            yield int(line)

# it = read_visits('my_number.txt')
# percentage = normalize(it)
# print(percentages)
```

```
# >>>
# []
```

- the output of the above statement is [] because an iterator produces it results only a signle time
- if you iterate over an iterator or a generator that has already raised a `StopIteration` exception, you wont get any results the second time around

- to bypass this problem just create a copy of the iterator and iterater through that
- `iterator_copy = list(nums)`
- downside is that an iterator can be extreemely large and could cause the program to run out of memeory and crash
- a workaround is to accept a function that returns a new iterator each time its called

[32]:
```python
def normalize_func(get_iter):
    total = sum(get_iter()) # New iterator
    result = []
    for value in get_iter(): # New iterator
        percent = 100 * value / total
        result.append(precent)
    return result

# path = 'my_numbers.txt'
# percentages = normalize_func(lambda: read_visits(path))
# print(percentages)
# assert sum(percentages) == 100.0
```

- to use `normaize_func` we have to pass it a lambda expression that calls the generator to produce a new iterator each time
- the lambda function is clumsy and thus a better way to do this is to use the `iterator protocol`

- the `iterator protocol` is how python `for` loops and related expressions traverse the contents of a container type
- when python sees a statement like `for x in foo` it acutally calls `iter(foo)`
- the `iter` built-in function calls the `foo.__iterator__` special method in turn
- the `__iter__` methoth must return and iterator object (which itself implements the `__next__` special method)
- then, the `for` loop repeatedly calls the `next` built-in function on the iterator object untill its exhausted (indicated by raising a `StopIteration` exception)

- all that to say that you can achieve all of this behavior for you classes by implementing the `__iter__` method as a generator

[39]:
```python
class ReadVisits:
    def __init__(self, data_path):
        self.data_path = data_path

    def __iter__(self):
```

```
        with open(self.data_path) as f:
            for line in f:
                yield int(line)

# visits = ReadVisits(path)
# percentages = normalize(visits)
# print(percentages)
# assert sum(percentages) == 100.0
```

- this works because the `sum` method in `normalize` calls `ReadVisits.__iter__` to allocate new iterator objects
- the `for` loop to normalize the numbers also calls `__iter__` to allocate a second iterator object
- each of those iterators will be advanced and exhausted independently, ensuring that each unique iterator sees all of the input data values
- the donside of this approach is that it reads the input data multiple times

- now that you know how containers like `ReadVisits` work, you can write your functions and methods to ensure that parameters arent just iterators
- the protocal states that when an iterator is passed to the `iter` built-in function, `iter` returns the iterator itself
- in contrast, when a container is passed to the `iter`, a new iterator object is returned each time

Things To Remember

- beware of functions and methods that iterate over input arguments multiple time. If these arguments are iterators, you may see strange behavior and missing values
- python iterator protocol defines how containers and iterators interact with `iter` and `next` built-in functions `for` loops and related expressions
- you can easily define your own iterable container type by implementing the `__iter__` method as a generator
- you can detect that a value is an iterator (instead of a container) if calling `iter` on its produces the same value as you passed in. Alternatively you can use the `isinstance` built-in function along with the `collections.abc.Iterator` class

0.7 Item 32: Consider Generator Expressions for Large List Comprehensions

- the problem with `list comprehensions` is that they may create new `list` instances containing one item for each value in the input sequence
- for large inputs, this could pose a problem
- if I wanted to read a file and return the number of characters on each line, it would require holding the lenght of every line of the file in memory

- to solve this problem, python provides `generator expressions`, which are a generalization of list comprehensions and generators
- generator expressions don;t materialize the whole output sequence when they're run
- instead, generator expressions evaluate to an iterator that yeilds one itme at a time from the expression

[38]:
```
def compare_comprehensions():
    value = [len(x) for x in open('my_file.txt')]

    it = (len(x) for x in open('my_file.txt'))
    print(it)
```

- the returned iterator can be advanced one step at a time to produce the next output from the generator expression, as needed
- another major benefit of generator expressions is that they can be combined

[37]:
```
def combine_expressions():
    it = (len(x) for x in open('my_file.txt'))
    roots = ((x, x**0.5) for x in it)
```

- each time I advance this iterator, it also advances the interior iterator, creating a domino effect of looping, evaluating expressions and passing around inputs and outputs, all while being as memory effcient as possible
- the only gotcha is that iterators returned by generators expressions are stateful so you must be careful not to use these iterators more than once
- Generator expressions execute very quickly when chained together and are memory efficient

0.8 Item 33: Compose Multiple Generators with `yield from`

- lets say I have a graphical program that's using generators to animate the movement of images onscreen
- to get the visual effect, I need the images to move quickly at first, puase and then continue moving at a slower pace

[40]:
```
def move(period, speed):
    for _ in range(period):
        yield speed

def pause(delay):
    for _ in range(delay):
        yield 0
```

- to create the final animation, I need to combine `move` and `pause` together to produce a single sequence of onscreen deltas
- this is done by calling a generator for each step of the animation, iterating over each generator in turn and then yielding the deltas from all of them in sequence

[41]:
```
def animate():
    for delta in move(4, 5.0):
        yield delta
    for delta in pause(3):
        yield delta
    for delta in move(2, 3.0):
        yield delta
```

- now we can render those deltas onscreen as they're produced by the single `animation` generator

[43]:
```
def render(delta):
    print(f'Delta: {delta:.1f}')
    # move the image on screen

def run(func):
    for delta in func():
        render(delta)

run(animate)
```

```
Delta: 5.0
Delta: 5.0
Delta: 5.0
Delta: 5.0
Delta: 0.0
Delta: 0.0
Delta: 0.0
Delta: 3.0
Delta: 3.0
```

- the problem with the code above is the repetitive nature of the `animate` function
- the reduncancy of the `for` statements and `yield` expressions for each generator add noise and reduces readability

- the solution is to use the `yield from` expression
- this advanced generator feature allows you to `yeild` all values from a nested generator before returning control to the parent generator

[44]:
```
def animate_composed():
    yield from move(4, 5.0)
    yield from pause(3)
    yield from move(2, 3.0)

run(animate_composed)
```

```
Delta: 5.0
Delta: 5.0
Delta: 5.0
Delta: 5.0
Delta: 0.0
Delta: 0.0
Delta: 0.0
Delta: 3.0
Delta: 3.0
```

- `yeild from` causes the Python interperter to handle the nested `for` loop and `yield` expression boilerplate for you, which results in better performance

0.9 Item 34: Avoid Injecting Data into Generators with `send`

- `yield` expressions provide generator functions with a simple way to produce an iterable series of output values
- this channel appears to be unidirectional, meaning there's no immediately obvious way to simultaneously stream data in and out of a generator ad it runs
- having bidirectional communication could be valuable

- let say I am writing a program to transmit signals using a software-defined radio
- the code can work fine for producing basic waveforms but it cnt be used to constantly vary the amplitude of the wave based on a seprate input
- we need a way to modulate the amplitude on each iteration of the generator
- python generators support the `send` method, which upgrades `yield` expressions into a two-way channel
- the `send` method can be used to provide streaming inputs to a generator at the same time its yeilding outputs
- normally when iterating a generator, the value of the yield expression is `None`

[46]:
```
def my_generator():
    received = yield 1
    print(f'received = {received}')

it = iter(my_generator())
output = next(it) # get first generator output
print(f'output = {output}')

try:
    next(it) # Run generator unitl it exits
except StopIteration:
    pass
```

```
output = 1
received = None
```

- when we call the `send` method instead of iterating the generator with a for loop or the next built-in function, the supplied parameter becomes the value of the `yield` expression when the generator is resumed
- howerver, when the generator first starts, a `yield` expression has not been encountered yet, so the only valid value for calling `send` initially is `None`

[49]:
```
it = iter(my_generator())
output = it.send(None) # Get first generator output
print(f'output = {output}')

try:
    it.send('hello!') # send value into iterator output
except StopIteration:
    pass
```

```
output = 1
```

```
received = hello!
```

[54]:
```python
import math

def wave_modulating(steps):
    step_size = 2 * math.pi / steps
    amplitude = yield # Receive initial amplitude
    for step in range(steps):
        radians = step * step_size
        fraction = math.sin(radians)
        output = amplitude * fraction
        amplitude = yield output # Receive next amplitude

def run_modulating(it):
    amplitudes = [ None, 7, 7, 7, 2, 2, 2, 2, 10, 10, 10, 10, 10]
    for amplitude in amplitudes:
        output = it.send(amplitude)
        transmit(output)

# run_modulating(wave_modulating(12))
```

- just avoid the `send` method entirry due to it confusing `None` Relations
- the easiest solution is to pass an iterator into the `wave` function
- the iterator should return an input amplitude each time the `next` built-in function is called on it

0.10 Item 35: Avoid Causing State Transitions in Generators with `throw`

- the `throw` is an andvanced generator feature for re-raising `Exception` instanced within generator functions
- when the method is called, the `next` occurrence of `yeild` expression `re-raises` the provided Exception instance after its output is received instead of continuing normally

[55]:
```python
class MyError(Exception):
    pass

def my_generator():
    yield 1
    yield 2
    yield 3

it = my_generator()
print(next(it)) # Yield 1
print(next(it)) # Yield 2
print(it.throw(MyError('test error')))
```

```
1
2
```

```
---------------------------------------------------------------------------
MyError                                   Traceback (most recent call last)
<ipython-input-55-8301c72fcd63> in <module>
     10 print(next(it)) # Yield 1
     11 print(next(it)) # Yield 2
---> 12 print(it.throw(MyError('test error')))

<ipython-input-55-8301c72fcd63> in my_generator()
      4 def my_generator():
      5     yield 1
----> 6     yield 2
      7     yield 3
      8

MyError: test error
```

- when you call `throw`, the generator function may catch the injected exception with a standard `try/except` compound statement that surrounds the last `yield` expression that was executed

[58]:
```python
def my_generator():
    yield 1

    try:
        yield 2
    except MyError:
        print('Got MyError!')
    else:
        yield 3

    yield 4

it = my_generator()
print(next(it)) # Yield 1
print(next(it)) # Yield 2
print(it.throw(MyError('test error')))
```

```
1
2
Got MyError!
4
```

- this functionality provides a two-way communication channel between a generator and its caller that can be useful in certain situations
- imagine if you want to write a program with a timer that supports sporadic resents

[60]:
```python
class Reset(Exception):
    pass
```

```python
def timer(period):
    current = period
    while current:
        current -= 1
        try:
            yield current
        except Reset:
            current = period
```

- in the code, whenever the `Reset` exception is raised by the `yield` expression, the counter resets itself to its orginal period

- a simpler approach is to implement this functionality is to define a statrful closure using an iterable container object

[63]:
```python
class Timer:
    def __init__(self, period):
        self.current = period
        self.period = period

    def reset(self):
        self.current = self.period

    def __iter__(self):
        while self.current:
            self.current -= 1
            yield self.current

def run():
    timer = Timer(4)
    for current in timer:
        if check_for_reset():
            timer.reset()
        announce(current)
```

- the `throw` method can be used to re-raise exceptions within generators at the position of the most recently ececuted `yield` expression
- using `throw` harms readability because it requires additional nesting and boilerplate in order to raise and catch exceptions
- a better way to provide exceptional behavior in generators is to use a class that implements the `__iter__` method along with methods to cause exceptional state transitions

0.11 Item 36: Consider `itertools` for working with Iterators and Generators

- `itertools` built-in module contains a large number of functions that are useful for organizing and interacting with iterators
- `import itertools`

- whenever you find yourself dealing with tricky iteration code, its worth looking at itertools documentation

[66]:
```
import itertools
```

0.11.1 Linking Iterator Together

chain: - use `chain` to combine multiple iterators into a single sequential iterator

[67]:
```
it = itertools.chain([1, 2, 3], [4, 5, 6])
print(list(it))
```

```
[1, 2, 3, 4, 5, 6]
```

repeat: - use `repeat` to output a single value forever, or use the second parameter to specify a maximum number of times

[68]:
```
it = itertools.repeat('hello', 3)
print(list(it))
```

```
['hello', 'hello', 'hello']
```

cycle: - use `cycle` to repeat an iterators items forever

[69]:
```
it = itertools.cycle([1, 2])
result = [next(it) for _ in range(10)]
print(result)
```

```
[1, 2, 1, 2, 1, 2, 1, 2, 1, 2]
```

tee: - use `tee` to split a single iterator into the number of parallel iterators specified by the second parameter - the memory usage of this function will grow if the iterators `dont` progress at the same speed since buffering will be required to enqueue the pending items

[72]:
```
it1, it2, it3 = itertools.tee(['first', 'second'], 3)

print(list(it1))
print(list(it2))
print(list(it3))
```

```
['first', 'second']
['first', 'second']
['first', 'second']
```

zip longest: - this variant of the `zip` built-in function returns a placeholder value when an iterator is exhausted, which may happen if iterators have different lenght

[75]:
```
keys = ['one', 'two', 'three']
values = [1, 2]

normal = list(zip(keys, values))
print('zip: ', normal)
```

```
it = itertools.zip_longest(keys, values, fillvalue='nope')
longest = list(it)
print('zip_longest', longest)
```

```
zip:  [('one', 1), ('two', 2)]
zip_longest [('one', 1), ('two', 2), ('three', 'nope')]
```

0.11.2 Filtering Items from an Iterator

- the itertools built-in module includes a number of functions for filtering items from an iterator

islice: - use `islice` to slice an iterator by numerical indexes without copying - you can specify the end, start and end, or start, end, and step sizes, and the behavior is simmilar to that of standard sequence slicing and striding

[77]:
```
values = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

first_five = itertools.islice(values, 5)
print('First five: ', list(first_five))

middle_odds = itertools.islice(values, 2, 8, 2)
print('Middle odds:', list(middle_odds))
```

```
First five:  [1, 2, 3, 4, 5]
Middle odds: [3, 5, 7]
```

takewhile: - `takewhile` returns items from an iterator untill a predicate function returns `False` for an item

[78]:
```
values = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
less_than_seven = lambda x: x < 7
it = itertools.takewhile(less_than_seven, values)
print(list(it))
```

```
[1, 2, 3, 4, 5, 6]
```

dropwhile: - `dropwhile` is the oppsite of `takewhile`, skips items from an iterator untill the predicate function returns `True` for the first time

[79]:
```
values = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
less_than_seven = lambda x: x < 7
it = itertools.dropwhile(less_than_seven, values)
print(list(it))
```

```
[7, 8, 9, 10]
```

filterfalse: - `filterfalse`, which is the oppsite of `filter` - it returns all items from an iterator where a predicate function feturns `False`

[82]:
```
values = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
evens = lambda x: x % 2 == 0

filter_result = filter(evens, values)
print('Filter:        ', list(filter_result))

filter_false_result = itertools.filterfalse(evens, values)
print('Filter false:', list(filter_false_result))
```

```
Filter:         [2, 4, 6, 8, 10]
Filter false: [1, 3, 5, 7, 9]
```

0.11.3 Producing Combinations of Items from Iterators

- `itertools` built-in module includes a number of functions for producing combinations of items from iterators

accumulate: - `accumulate` folds an item from an iterator into a running value by applying a function that takes two parameters - it outputs the current `accumulated result` for each input value

[83]:
```
values = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
sum_reduce = itertools.accumulate(values)
print('Sum:   ', list(sum_reduce))

def sum_modulo_20(first, second):
    output = first + second
    return output % 20

modulo_reduce = itertools.accumulate(values, sum_modulo_20)
print('Modulo', list(modulo_reduce))
```

```
Sum:    [1, 3, 6, 10, 15, 21, 28, 36, 45, 55]
Modulo [1, 3, 6, 10, 15, 1, 8, 16, 5, 15]
```

- this is bascially the same as `reduce` functions from `functools` built-in module but withou outputs `yielded` one step at a time
- by default it sums the input if no binary function is specified

product: - `product` returns the Cartesian product of items from one or more iterators - which is a nice alternative to using deeply nested list comprehensions - simmilar to combination with a fixed `N`

[84]:
```
single = itertools.product([1, 2], repeat=2)
print('Single: ', list(single))

multiple = itertools.product([1, 2], ['a', 'b'])
print('Multiple: ', list(multiple))
```

```
Single:  [(1, 1), (1, 2), (2, 1), (2, 2)]
Multiple:  [(1, 'a'), (1, 'b'), (2, 'a'), (2, 'b')]
```

permutations: - `permutations` returns the unique ordered permutations of lenght `N` with item from an iterator

[86]:
```
it = itertools.permutations([1, 2, 3, 4], 2)
print(list(it))
```

```
[(1, 2), (1, 3), (1, 4), (2, 1), (2, 3), (2, 4), (3, 1), (3, 2), (3, 4), (4, 1),
(4, 2), (4, 3)]
```

combinations: - `combinations` returns an unordered combinations of lenght `N` with repeated items from an iterator

[90]:
```
it = itertools.combinations([1, 2, 3, 4], 2)
print(list(it))
```

```
[(1, 2), (1, 3), (1, 4), (2, 3), (2, 4), (3, 4)]
```

combinations_with_replacement: - `combinations_with_replacement` is the same as `combinations` but repeated values are allowed - basically permutations with a variable `N`

[91]:
```
it = itertools.combinations_with_replacement([1, 2, 3, 4], 2)
print(list(it))
```

```
[(1, 1), (1, 2), (1, 3), (1, 4), (2, 2), (2, 3), (2, 4), (3, 3), (3, 4), (4, 4)]
```

- The itertools functions fall into three main categories for working with iterators and generators: linking iterators together,