Iterators

Amit Kotlovski
Open-Source Consultancy
http://amitkot.com

Agenda

- List comprehension
- Generator expressions
- Generator functions
- Iteration tools

List comprehension

List comprehension

- A pythonic syntax for replacing `map()` and `filter()`

map

Runs a function on each item in the sequence

```
def times6(x):
    return x * 6

>>> data = [-2, -1, 0, 1, 2]

>>> map(times6, data)
[-12, -6, 0, 6, 12]

>>> map(lambda x: x ** 2, data)
[4, 1, 0, 1, 4]
```

filter

Filter a sequence for items matching the condition

```
def negative(x):
    return x < 0

>>> data = [-2, -1, 0, 1, 2]

>>> filter(negative, data)
[-2, -1]

>>> filter(lambda x: x % 2 == 0, data)
[-2, 0, 2]
```

Together

```
>>> data = [-2, -1, 0, 1, 2]
```

```
# filter + map
>>> filtered = filter(lambda x: x < 0, data)
>>> filtered_mapped = map(lambda x: x * 6, filtered)
>>> print(filtered_mapped)
[-12, -6]
```

```
# list comprehension
>>> list_comp = [x * 6 for x in data if x < 0]
>>> print(list_comp)
[-12, -6]
```

```python
# filter + map
filtered = filter(filter_func, data)
result = map(map_func, filtered)

# list comprehension
result = [
    map_func(x)
    for x in data
    if filter_func(x)
]

# for loop
result = []
for x in data:
    if filter_func(x):
        result.append(map_func(x))
```

Parallel list comprehension

Used for iterating over more than one list

```
>>> veg = ['eggplant', 'carrot', 'pepper']
>>> sauce = ['tehini', 'mustard', 'spicy']
>>> [(v, s) for v in veg for s in sauce]
[('eggplant', 'tehini'),
 ('eggplant', 'mustard'),
 ('eggplant', 'spicy'),
 ('carrot', 'tehini'),
 ('carrot', 'mustard'),
 ('carrot', 'spicy'),
 ('pepper', 'tehini'),
 ('pepper', 'mustard'),
 ('pepper', 'spicy')]
```

Dictionary comprehension

Use a similar syntax to construct a dictionary

```
with open('/usr/share/dict/words') as f:
    data = [word.strip() for word in f]

>>> res = {
        word[:5]: word[-5:]
        for word in data
        if len(word) > 23
    }

>>> print(res)
{'forma': 'ylate',
 'patho': 'gical',
 'scien': 'hical',
 'tetra': 'alein',
 'thyro': 'omize'}
```

Set comprehension

A similar syntax for creating sets

```
>>> s = {word[-3:] for word in data if len(word) > 23}
>>> print(s)
{‘ate’, 'cal', 'ein', 'ize'}
```

Q&A

https://secure.flickr.com/photos/21560098@N06/3836926854/

Generator expressions

Generator expressions

- List comprehensions are great for building a list of items
- But sometimes we don't need the actual list
- We just want to iterate over the items
- (Save time and memory)

A “lazy” list

- Remember `range()` and `xrange() [Python 2]`?
- Generator expressions are to list comprehension as `xrange()` is to `range()`
- A “lazy” list that can return the next item each time

Familiar syntax

Using parentheses () instead of brackets []

```
data = 'a small cat has jumped over my table'.split()
even_words_list = [word for word in data if len(word) % 2 == 0]
even_words_gen = (word for word in data if len(word) % 2 == 0)

>>> for item in even_words_list:
...     print(item)
jumped
over
my

>>> for item in even_words_gen:
...     print(item)
jumped
over
my
```

But...

```
>>> print(even_words_gen)
<generator object <genexpr> at 0x10dd950f0>

# and

>>> len(even_words_gen)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: object of type 'generator' has no len()
```

What's going on?

- List comprehension return a list
- Generator expressions return a generator

So, what is a generator?

- First lets talk a bit about iterators

What is an iterator?

- A mechanism for iterating over sequences

What's inside an Iterator?

- `__next__()` function for getting the next item (`next()` in Python 2)
  - Calling `__next__()` when there are no more items raises the `StopIteration` exception
- The `__iter__()` magic function that returns the iterator itself

```
data = [1, 2, 3]
li = iter(data)

>>> next(li)
1

>>> next(li)
2

>>> next(li)
3

>>> next(li)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
StopIteration
```

What happens in a for loop?

```
for item in obj:
    do_something(item)
```

Get iterator	`it = iter(obj)`
Get first item	`item = next(it)`
Get next item	`item = next(it)`
...	`...`
Get last item	`item = next(it)`
Get another item	`item = next(it) --> raise StopIteration()`

What happens in a for loop?

- The sequence's iterator is acquired by calling the `iter()` function (internally calling its `__iter__()` function)

- The sequence's items are provided by the iterator `iter` by calling `next(iter)` in each iteration (internally calling `__next__()`

- Until a `StopIteration` is raised

OK, so a generator is an iterator?

- Well, not exactly

- But since a generator implements the iterator protocol, it behaves the same in for loops

Back to generator expressions

- Similar to list comprehensions
- The list is “lazy” - not created in advance, each item calculated when `next()` is called
- More efficient (time, memory) for large data sets

Comparing run times

```
with open('/usr/share/dict/words') as f:
    data = [word.strip() for word in f]

>>> len(data)
235886

>>> %timeit [word for word in data]
8.02 ms ± 550 µs per loop (mean ± std. dev. of 7 runs,
100 loops each)

>>> %timeit (word for word in data)
389 ns ± 8.8 ns per loop (mean ± std. dev. of 7 runs,
1000000 loops each)

# 1 nanosecond = 1/1,000,000 millisecond
# --> creation time is about x20,000 faster
```

Comparing memory size

- How much memory will loading a 1 petabyte (1,000 terabyte) file require:

- With a list comprehension?

- With a generator expression?

Easily convert to list

```
>>> ge = (word.upper() for word in 'apple pie'.split())
>>> l = list(ge)
>>> print(l)
['APPLE', 'PIE']
```

Q&A

https://secure.flickr.com/photos/21560098@N06/3836926854/

Generator functions

Iterating

- To iterate over an object it must implement:
  - The iterator protocol, or
  - The `__getitem__()` magic function

__getitem__()

- Easy for objects that contain indexed items

Implementing a list wrapper with `__getitem__()`

```
class ListWrapper(object):
    def __init__(self, *elements):
        self.elements = elements
    def __getitem__(self, i):
        return self.elements[i]

>>> lw = ListWrapper(1, 22, 333)
>>> lwi = iter(lw)
>>> print(next(lwi))
1
>>> print(next(lwi))
22
>>> print(next(lwi))
333
>>> print(next(lwi))
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
StopIteration
```

Let's implement the iterator protocol

Iterator protocol

- implement `__iter__()` to return itself
- Implement `__next__()`
- Raise `StopIteration` when there are no more items

count iterator

```
class CountIterator(object):
    def __init__(self):
        self.curr = 0
    def __iter__(self):
        return self
    def __next__(self):
        res = self.curr
        self.curr += 1
        return res

>>> c = CountIterator()
>>> ci = iter(c)
>>> print(next(ci))
0
>>> print(next(ci))
1
>>> print(next(ci))
2
```

What to use?

- The `__getitem__()` approach is easy for indexed items
- The iterator protocol is implemented where there is logic involved

Generator function

- A convenient way to implement the iterator protocol
- When next() is called the function yields the next value in the iteration

```
def count_func():
    curr = 0
    while True:
        yield curr
        curr += 1
```

Trying it out

```
def count_func():
    curr = 0
    while True:
        yield curr
        curr += 1

c = count_func()
>>> print(next(c))
0
>>> print(next(c))
1
>>> print(next(c))
2
```

Q&A

https://secure.flickr.com/photos/21560098@N06/3836926854/

Iteration tools - `itertools`

itertools

- A collection of tools for creating, combining and manipulating iterators

- Inspired by APL, Haskell and SML languages

Infinite iterators

- Following are iterators that can be called infinitely and will keep on returning items

count()

Arguments	[start=0] [, step=1]

Return growing numbers starting from `start` with the given `step`

```
>>> c = count(8, 2)
>>> print(next(c))
8
>>> print(next(c))
10
>>> print(next(c))
12
```

`cycle()`

Arguments	iterable

Cycle through the elements in sequence `iterable`

```
>>> c = cycle(['cat', 'dog'])
>>> next(c)
'cat'
>>> next(c)
'dog'
>>> next(c)
'cat'
```

`repeat()`

Arguments	object[, times=None]

Repeatedly return object forever or given number of `times`

```
>>> r = repeat(3, times=2)
>>> next(r)
3
>>> next(r)
3
>>> next(r)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
StopIteration
```

Q&A

https://secure.flickr.com/photos/21560098@N06/3836926854/

Tools for one iterable

- Or the shortest of provided ones

chain()

Arguments	*iterables

Return all items from first iterable, then second iterable, until the last one ends

```
>>> mine = [34, 12]

>>> theirs = [99, 1, 3]

>>> c = itertools.chain(mine, theirs)
>>> print(c)
<itertools.chain at 0x10fbb0080>

>>> list(c)
[34, 12, 99, 1, 3]
```

`compress()`

Arguments	data, selectors

Filter elements from `data` according to the truth value of the corresponding elements in `selectors`

```
>>> c = compress('cat', [True, False, True])
>>> next(c)
'c'
>>> next(c)
't'
>>> next(c)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
StopIteration
```

`takewhile()`

Arguments	predicate, iterable

Return elements from `iterable` as long as `predicate` is true

```
>>> t = takewhile(lambda x: x > 0, [1, 2, -1, 2, 3])
>>> next(t)
1
>>> next(t)
2
>>> next(t)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
StopIteration
```

`dropwhile()`

Arguments	predicate, iterable

Drop elements from `iterable` as long as `predicate` is false, then start returning all elements from it

```
>>> d = dropwhile(lambda x: x > 0, [1, 2, -1, 2, 3])
>>> next(d)
-1
>>> next(d)
2
>>> next(d)
3
>>> next(d)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
StopIteration
```

`groupby()`

- Receive an iterable and an optional key function
  - Iterable is expected to be sorted by that key function
- Returns `(key, iterator)` tuple for groups of items sharing that key

groupby()

Arguments	iterable[, key=None]

```
>>> gb = groupby(['cat', 'bag', 'leg', 'peg', 'dog', 'hog', 'bog'],
                 lambda x: x[1])
>>> for k, v in gb:
...     print('key=({}), value=({})'.format(k, list(v)))
...
key=(a), value=(['cat', 'bag'])
key=(e), value=(['leg', 'peg'])
key=(o), value=(['dog', 'hog', 'bog'])
```

`groupby()`

- Result group iterators can be saved as lists
- Default key is the identity function
- Similar to Unix `uniq`

`filter()` (builtin)

Arguments	predicate, iterable

Return elements from `iterable` for which `predicate` is true. `None` predicate returns items that are True.

```
>>> f = ifilter(lambda x: x.islower(),
                'a HUGE strawberry'.split())
>>> next(f)
'a'
>>> next(f)
'strawberry'
>>> next(f)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
StopIteration
```

`filterfalse()`

Arguments	predicate, iterable

Return elements from `iterable` for which `predicate` is false. `None` predicate returns items that are False.

```
>>> f = ifilterfalse(lambda x: x.islower(),
        'a HUGE strawberry'.split())
>>> next(f)
'HUGE'
>>> next(f)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
StopIteration
```

`islice()`

Arguments	iterable, stop iterable, start, stop[, step]

Returns items from `iterable` from `start` until `stop` with optional `step`

```
>>> s = islice('abcdefghijk', 1, 7, 2)
>>> next(s)
'b'
>>> next(s)
'd'
>>> next(s)
'f'
>>> next(s)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
StopIteration
```

islice()

- Unlike regular slicing does not accept negative indices

`map()` (builtin)

Arguments	func, *iterables

Call `func` with arguments from each of the `iterables`

```
>>> import operator
>>> im = map(operator.mul, 'ABC', (1, 2, 3))
>>> next(im)
'A'
>>> next(im)
'BB'
>>> next(im)
'CCC'
>>> next(im)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
StopIteration
```

map() (builtin)

If `func` is None returns an iterable of tuples from the `iterables`

```
>>> im = map(None, 'ABC', (1, 2, 3))
>>> next(im)
('A', 1)
>>> next(im)
('B', 2)
>>> next(im)
('C', 3)
>>> next(im)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
StopIteration
```

`starmap()`

Arguments	func, iterable

Call `func` with arguments from the `iterable`

```
>>> import operator
>>> sm = starmap(operator.mul, [('A', 1), ('B', 2), ('C', 3)])
>>> next(sm)
'A'
>>> next(sm)
'BB'
>>> next(sm)
'CCC'
>>> next(sm)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
StopIteration
```

`tee()`

Arguments	iterable[, n=2]

Return `n` iterators for the provided `iterable`

```
>>> it1, it2 = tee(range(5))
>>> next(it1)  # first iterator
0
>>> next(it1)
1
>>> next(it1)
2
>>> next(it2)  # second iterator
0
>>> next(it2)
1
```

`zip()` (builtin)

Arguments	iterables

Return an iterator that aggregates items in the same position from all `iterables`, similar to `zip()`

```
>>> iz = zip('abc', [1, 2, 3])
>>> next(iz)
('a', 1)
>>> next(iz)
('b', 2)
>>> next(iz)
('c', 3)
>>> next(iz)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
StopIteration
```

```
>>> stuff = ['Electric Light', 'Microwave Oven',
...          'Canned Food']
>>> people = ['Thomas Edison', 'Percy Spencer',
...           'Nicolas Appert']
>>> inventions = zip(people, stuff)
>>> next(inventions)
('Thomas Edison', 'Electric Light')
>>> next(inventions)
('Percy Spencer', 'Microwave Oven')
>>> next(inventions)
('Nicolas Appert', 'Canned Food')
>>> next(inventions)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
StopIteration
```

`zip_longest()`

Arguments	iterables[, fillvalue=None]

Similar to `zip()`, filling the shorter `iterables` with `fillvalue` items

```
>>> iz = zip('a', [1, 2, 3])
>>> next(iz)
('a', 1)
>>> next(iz)
Traceback (most recent call
last):
  File "<stdin>", line 1, in
<module>
StopIteration
```

```
>>> from itertools import zip_longest
>>> iz = zip_longest('a', [1, 2, 3])
>>> next(iz)
('a', 1)
>>> next(iz)
(None, 2)
>>> next(iz)
(None, 3)
>>> next(iz)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
StopIteration
```

Python 2 --> Python 3

- In Python 2 some functions returned a list and had an similar function in itertools for returning an iterator
- In Python 3 these functions return an iterator

Returns	Python 2	Python 3
`list`	`map(...)`	`list(map(...))`
`list`	`filter(...)`	`list(filter(...))`
`list`	`zip(...)`	`list(zip(...))`
`list`	`range(...)`	`list(range(...))`
`iterator`	`itertools.imap(...)`	`map(...)`
`iterator`	`itertools.ifilter(...)`	`filter(...)`
`iterator`	`itertools.izip(...)`	`zip(...)`
`iterator`	`xrange(...)`	`range(...)`

Q&A

https://secure.flickr.com/photos/21560098@N06/3836926854/

Combinatoric iterators

product()

Arguments	*iterables[, repeat]

Cartesian product of input iterables

```
>>> p = product('abc', '123')
>>> list(p)
[('a', '1'), ('a', '2'), ('a', '3'),
 ('b', '1'), ('b', '2'), ('b', '3'),
 ('c', '1'), ('c', '2'), ('c', '3')]
```

permutations()

Arguments	iterable[, r=len(iterable)]

Return `r` length permutations of the items in `iterable`

```
>>> p = permutations('abc', 2)
>>> list(p)
[('a', 'b'), ('a', 'c'), ('b', 'a'), ('b', ‘c'),
 ('c', 'a'), ('c', 'b')]
```

permutations()

Permutations are based on the items' position (not their value)

```
>>> list(permutations('aba'))
[('a', 'b', 'a'), ('a', 'a', 'b'), ('b', 'a', 'a'),
 ('b', 'a', 'a'), ('a', 'a', 'b'), ('a', 'b', 'a')]
```

`combinations()`

Arguments	iterable, r

Return `r` length combinations of `iterable`

```
>>> list(combinations('abc', 2))
[('a', 'b'), ('a', 'c'), ('b', 'c')]
```

`combinations_with_replacement()`

Arguments	iterable, r

Return `r` length combinations of `iterable`, allowing individual elements to be repeated more than once

```
>>> list(combinations_with_replacement('abc', 2))
[('a', 'a'), ('a', 'b'), ('a', 'c'), ('b', ‘b'),
 ('b', 'c'), ('c', 'c')]
```

Q&A

https://secure.flickr.com/photos/21560098@N06/3836926854/

Summary

- List comprehension for simple list manipulation
- Generator expressions for handling streams of data
- Generator functions for exposing iterators
- Itertools provide many functions for iterating over data

QUESTION
EVERYTHING

Thanks!

twitter @amitkot
www amitkot.com