Iteration and Iterables

Austin Bingham
COFOUNDER - SIXTY NORTH

@austin_bingham

Robert Smallshire
COFOUNDER - SIXTY NORTH

@robsmallshire

Overview

Comprehensions

- Creating familiar objects
- Creating new kinds of objects
- Filtering

Low-level iterable API

- Iterators
- Exceptions in iteration

Overview

Generator functions

- The `yield` keyword
- Statefulness, laziness, and infinite sequences
- Generator expressions

Iterations tools

Comprehensions

Concise syntax for describing lists, sets, and dictionaries.

Readable and expressive.

Close to natural language.

List Comprehensions

```
>>> words = "Why sometimes I have believed as many as six impossible things befo
re breakfast".split()
>>> words
['Why', 'sometimes', 'I', 'have', 'believed', 'as', 'many', 'as', 'six', 'imposs
ible', 'things', 'before', 'breakfast']
>>> [len(word) for word in words]
[3, 9, 1, 4, 8, 2, 4, 2, 3, 10, 6, 6, 9]
>>>
```

List Comprehension Syntax

```
[ expr(item) for item in iterable ]
```

Equivalent Syntax

```
>>> lengths = []
>>> for word in words:
...     lengths.append(len(word))
...
>>> lengths
[3, 9, 1, 4, 8, 2, 4, 2, 3, 10, 6, 6, 9]
>>> from math import factorial
>>> f = [len(str(factorial(x))) for x in range(20)]
>>> f
[1, 1, 1, 1, 2, 3, 3, 4, 5, 6, 7, 8, 9, 10, 11, 13, 14, 15, 16, 18]
>>> type(f)
<class 'list'>
>>>
```

The expression producing the new list's elements can be any Python expression.

Set Comprehensions

Set Comprehensions

```
>>> from math import factorial
>>> s = {len(str(factorial(x))) for x in range(20)}
>>> type(s)
<class 'set'>
>>> print(s)
{1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 13, 14, 15, 16, 18}
>>>
```

Dict Comprehensions

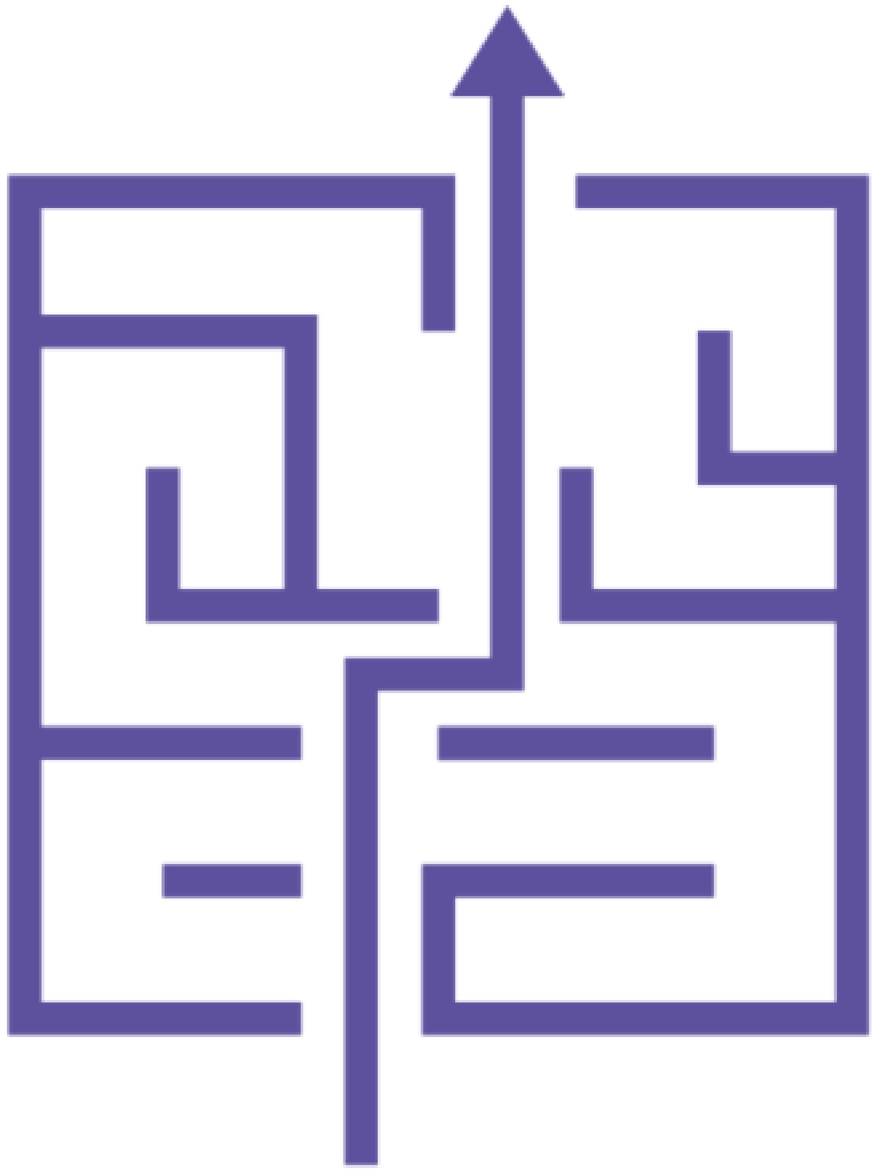
```
{
    key_expr(item): value_expr(item)
    for item in iterable
}
```

Dict Comprehensions

```
>>> country_to_capital = { 'United Kingdom': 'London',
...                        'Brazil': 'Brasília',
...                        'Morocco': 'Rabat',
...                        'Sweden': 'Stockholm' }
>>> capital_to_country = {capital: country for country, capital in country_to_ca
pital.items()}
>>> from pprint import pprint as pp
>>> pp(capital_to_country)
{'Brasília': 'Brazil',
 'London': 'United Kingdom',
 'Rabat': 'Morocco',
 'Stockholm': 'Sweden'}
>>> words = ["hi", "hello", "foxtrot", "hotel"]
>>> { x[0]: x for x in words }
{'h': 'hotel', 'f': 'foxtrot'}
>>>
```

Dictionary comprehensions don't work directly on `dict` sources.

Use `dict.items()` to get keys and values from `dict` sources.



Comprehension expressions can be arbitrarily complex

Avoid excessive complexity

Put complex expressions in separate functions for readability

Complex Expressions

```
>>> import os
>>> import glob
>>> file_sizes = {os.path.realpath(p): os.stat(p).st_size for p in glob.glob('*.
py')}
>>> pp(file_sizes)
{'/core-python/script.py': 649,
 '/core-python/session.py': 273}
```

Filtering Comprehensions

Filtering Comprehensions

```
>>> from math import sqrt
>>> def is_prime(x):
...     if x < 2:
...         return False
...     for i in range(2, int(sqrt(x)) + 1):
...         if x % i == 0:
...             return False
...     return True
...
>>> [x for x in range(101) if is_prime(x)]
[2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47, 53, 59, 61, 67, 71, 73,
 79, 83, 89, 97]
>>> prime_square_divisors = {x*x: (1, x, x*x) for x in range(20) if is_prime(x)}
>>> pp(prime_square_divisors)
{4: (1, 2, 4),
 9: (1, 3, 9),
 25: (1, 5, 25),
 49: (1, 7, 49),
 121: (1, 11, 121),
 169: (1, 13, 169),
 289: (1, 17, 289),
 361: (1, 19, 361)}
>>>
```

Moment of Zen

Simple is better than complex

Code is written once
But read over and over
Fewer is clearer

Comprehensions should normally have no side-effects.

Iteration Protocols

Iteration Protocols

iterable

Can be passed to `iter()` to produce an *iterator*

iterator

Can be passed to `next()` to get the next value in the sequence

Iteration Protocols

```
>>> iterable = ['Spring', 'Summer', 'Autumn', 'Winter']
>>> iterator = iter(iterable)
>>> next(iterator)
'Spring'
>>> next(iterator)
'Summer'
>>> next(iterator)
'Autumn'
>>> next(iterator)
'Winter'
>>> next(iterator)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
StopIteration
>>>
```

Stopping Iteration with an Exception

A single end

Sequences only have on ending, after all, so reaching it is exceptional

Infinite sequences

Finding the end of an infinite sequence would be truly exceptional

Iteration Protocols

```
>>> def first(iterable):
...     iterator = iter(iterable)
...     try:
...         return next(iterator)
...     except StopIteration:
...         raise ValueError("iterable is empty")
...
>>> first(["1st", "2nd", "3rd"])
'1st'
>>> first({"1st", "2nd", "3rd"})
'2nd'
>>> first(set())
Traceback (most recent call last):
  File "<stdin>", line 4, in first
StopIteration

During handling of the above exception, another exception occurred:

Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 6, in first
ValueError: iterable is empty
>>>
```

Generator Functions

Generator Functions

Iterables defined by functions

Lazy evaluation

Can model sequences with no definite end

Composable into pipelines

`yield`

Generator functions must include at least one `yield` statement.

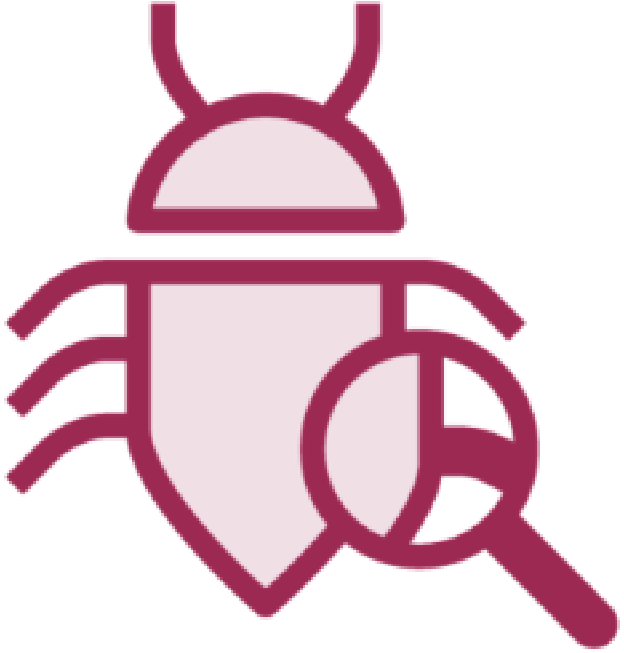
They may also include `return` statements.

Generator Functions

```
...     print("About to yield 2")
...     yield 2
...     print("About to yield 4")
...     yield 4
...     print("About to yield 6")
...     yield 6
...     print("About to return")
...
>>> g = gen246()
>>> next(g)
About to yield 2
2
>>> next(g)
About to yield 4
4
>>> next(g)
About to yield 6
6
>>> next(g)
About to return
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
StopIteration
>>>
```

Maintaining State in Generators

Graphical Debugger



Control flow is easier to see in a graphical debugger

This example uses PyCharm

`continue`

Finish current loop iteration and begin the next iteration immediately.

Lazy computation can result in complex flow control. Forced evaluation can simplify things during development.

Laziness and the Infinite

Generators only do enough work to produce requested data.

This allows generators to model infinite (or just very large) sequences.

Examples of such sequences are:

- **Sensor readings**
- **Mathematical sequences**
- **Contents of large files**

Laziness and the Infinite

```
>>> def lucas():
...     yield 2
...     a = 2
...     b = 1
...     while True:
...         yield b
...         a, b = b, a + b
...
>>> for x in lucas():
...     print(x)
...
2
1
3
4
7
11
18

KeyboardInterrupt
>>>
```

Generator Expressions

Generator Expressions

```
(expr(item) for item in iterable)
```

Generator Expressions

```
>>> million_squares = (x*x for x in range(1, 1000001))
>>> million_squares
<generator object <genexpr> at 0x1032cd450>
>>> list(million_squares)[-10:]
[999982000081, 999984000064, 999986000049, 999988000036, 999990000025, 999992000
016, 999994000009, 999996000004, 999998000001, 1000000000000]
>>> list(million_squares)
[]
>>>
```

To recreate a generator from a generator expression, you must execute the expression again.

Generator Expressions

```
>>> sum(x*x for x in range(1, 10000001))
333333383333335000000
>>>
```

Optional Parentheses

One set of parentheses

```
func((expr(item) for item in iterable))
```

These are still allowed

Generator Expressions

```
>>> sum(x for x in range(1001) if is_prime(x))
76127
>>>
```

Iteration Tools

Batteries Included

Python provides a powerful vocabulary for working with iterators

These include the familiar `enumerate()` and `sum()`

The `itertools` module provides many more

itertools.islice()

Perform lazy slicing of any iterator.

```
from itertools import islice
islice(all_primes, 1000)
```

`itertools.count()`

An unbounded arithmetic sequence of integers.

islice and count

```
>>> from itertools import count, islice
>>> thousand_primes = islice((x for x in count() if is_prime(x)), 1000)
>>> thousand_primes
<itertools.islice object at 0x10a0d8530>
>>> list(thousand_primes)[-10:]
[7841, 7853, 7867, 7873, 7877, 7879, 7883, 7901, 7907, 7919]
>>> sum(islice((x for x in count() if is_prime(x)), 1000))
3682913
>>>
```

Boolean Aggregation

`any()`	`all()`
Determines if any elements in a series are true	Determines if all elements in a series are true

any and all

```
>>> any([False, False, True])
True
>>> all([False, False, True])
False
>>> any(is_prime(x) for x in range(1328, 1361))
False
>>> all(name == name.title() for name in ['London', 'Paris', 'Tokyo', 'New York'
, 'Sydney', 'Kuala Lumpur'])
True
>>>
```

`zip()`

Synchronize iteration across two or more iterables.

zip

```
min = 11.0, max=22.0, average=18.0
min = 12.0, max=22.0, average=18.7
min = 10.0, max=23.0, average=18.3
min =  9.0, max=22.0, average=17.3
min =  8.0, max=20.0, average=15.7
min =  8.0, max=18.0, average=14.3
>>> from itertools import chain
>>> temperatures = chain(sunday, monday, tuesday)
>>> all(t > 0 for t in temperatures)
True
>>> for x in (p for p in lucas() if is_prime(p)):
...     print(x)
...
2
3
7
11
29
47
199
521

KeyboardInterrupt
>>>
```

Summary

Comprehensions for `list`, `set`, and `dict`

Comprehensions use an input iterable and an optional predicate

Iterable objects can be iterated item by item

Use `iter()` to get an iterator from an iterable object

Use `next()` to get the next item from an iterable

Iterators raise `StopIteration` when they're exhausted

Summary

Generator functions describe sequences imperatively

Generator functions contain at least one `yield`

Generators are iterators

Each call to a generator function produces a new generator

Generators maintain explicit internal state

Generators yield values lazily

Generator expressions are a type of comprehension that creates generators

Summary

Built-in iteration tools include `sum()`, `any()`, and `zip()`

The `itertools` module includes many other tools for iteration