Functional-style Tools

Austin Bingham
COFOUNDER - SIXTY NORTH
@austin_bingham

Robert Smallshire
COFOUNDER - SIXTY NORTH
@robsmallshire

Overview

Functional-style programming in Python

`map()`

`filter()`

`functools.reduce()`

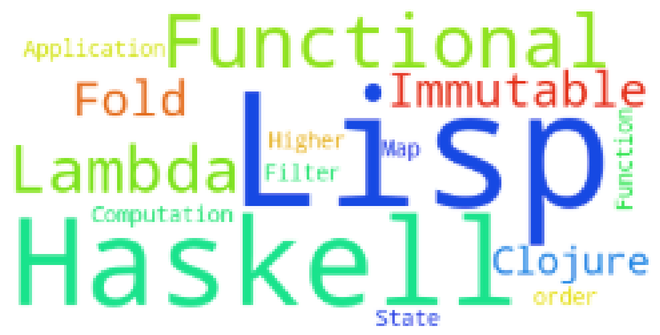
Combining these tools

Python's concept of iteration is simple and abstract

This allows us to use tools with an equally high level of abstraction

Python provides a number of "building block" functions

Functional Programming



Many of these ideas come from the functional programming community

They can be very useful and can be a great way to express certain computations

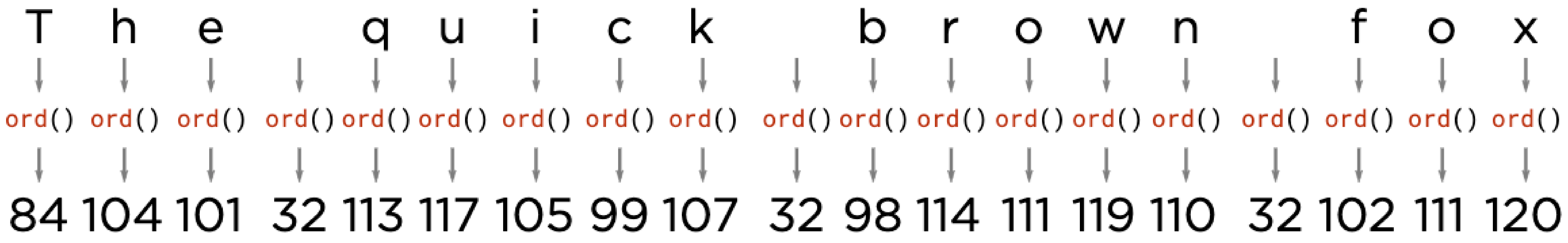
map()

Calls a function for the elements in a sequence, producing a new sequence with the return values

It "maps" a function over a sequence

map()

```
map(ord, 'The quick brown fox')
```


T h e q u i c k b r o w n f o x
ord() ord() ord() ord() ord() ord() ord() ord() ord() ord() ord() ord() ord() ord() ord() ord() ord() ord() ord()
84 104 101 32 113 117 105 99 107 32 98 114 111 119 110 32 102 111 120

map()

```
>>> map(ord, 'The quick brown fox')
<map object at 0x102ed20d0>
>>>
```

Map() Is Lazy

`map()` will not call its function or access its iterables until they're needed for output

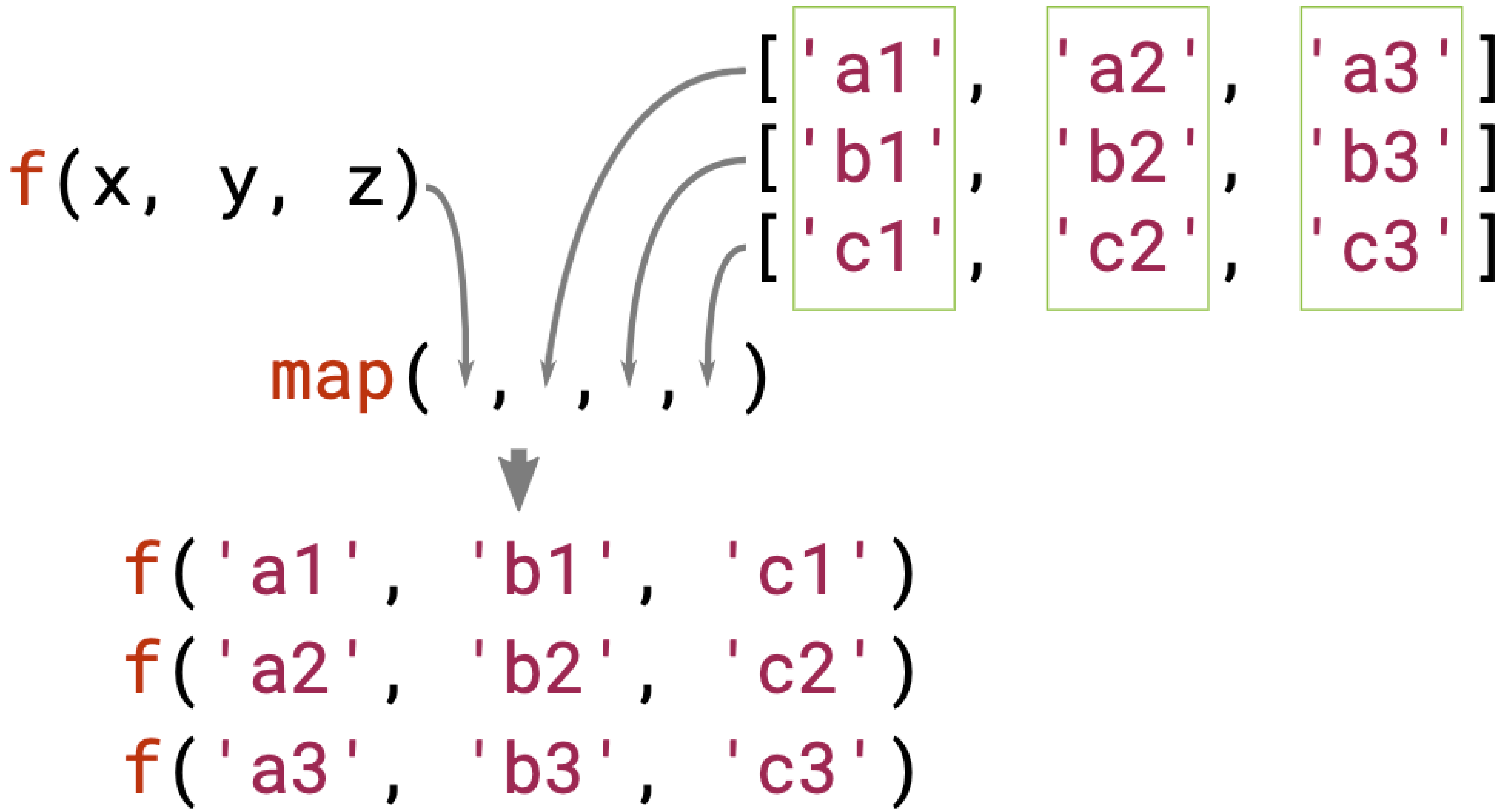
A map object is itself iterable; iterate over it to produce output

Tracing `map()`

```
111, 120]
>>> for o in map(ord, 'The quick brown fox'):
...     print(o)
...
84
104
101
32
113
117
105
99
107
32
98
114
111
119
110
32
102
111
120
>>>
```

`map()` can be used with as many input sequences as your mapped function needs.

map() with Multiple Iterables



map() with Multiple Iterables

```
>>> sizes = ['small', 'medium', 'large']
>>> colors = ['lavender', 'teal', 'burnt orange']
>>> animals = ['koala', 'platypus', 'salamander']
>>> def combine(size, color, animal):
...     return '{} {} {}'.format(size, color, animal)
...
>>> list(map(combine, sizes, colors, animals))
['small lavender koala', 'medium teal platypus', 'large burnt orange salamander'
]
>>> def combine(quantity, size, color, animal):
...     return '{} x {} {} {}'.format(quantity, size, color, animal)
...
>>> import itertools
>>> list(map(combine, itertools.count(), sizes, colors, animals))
['0 x small lavender koala', '1 x medium teal platypus', '2 x large burnt orange
 salamander']
>>>
```

map() and Comprehensions

```
>>> [str(i) for i in range(5)]
['0', '1', '2', '3', '4']
>>> list(map(str, range(5)))
['0', '1', '2', '3', '4']
>>> i = (str(i) for i in range(5))
>>> list(i)
['0', '1', '2', '3', '4']
>>> i = map(str, range(5))
>>> list(i)
['0', '1', '2', '3', '4']
>>>
```

`map()` vs. Comprehensions

Performance

Neither `map()` nor comprehensions are necessarily faster than the other.

Readability

Some people find one form more readable than the other.

Context

The choice between the two will often depend on your specific context.

`filter()`

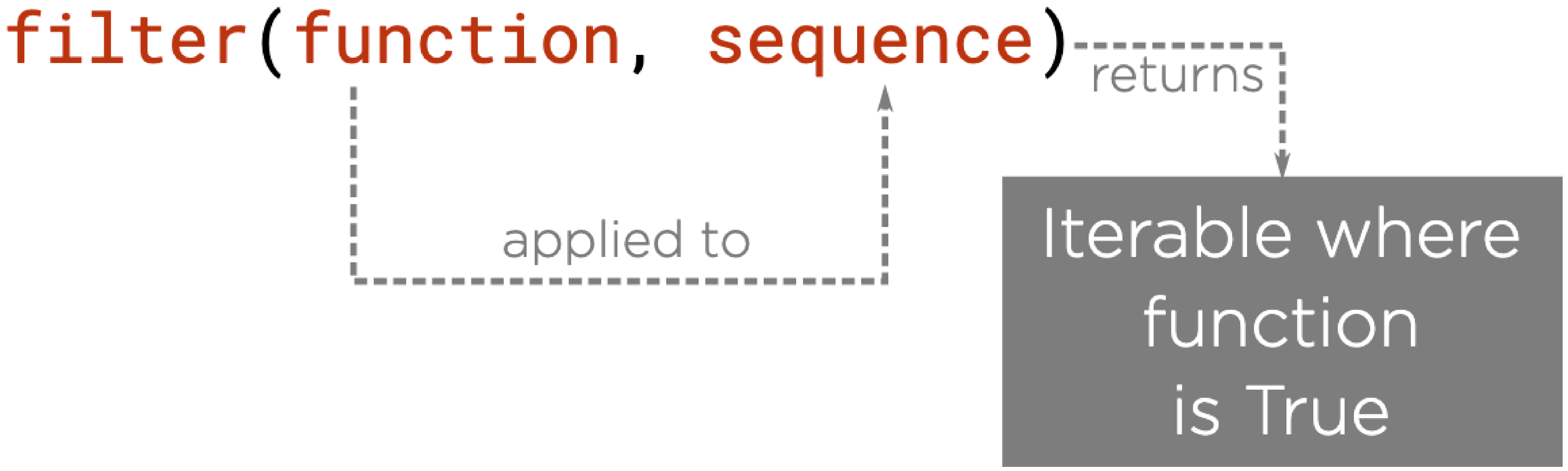
Removes elements from a sequence which don't meet some criteria

Applies a predicate function to each element

Produces its results lazily

Only accepts a single input sequence, and the function must accept only one argument

filter()


filter(function, sequence)
returns
applied to
Iterable where function is True

filter()

```
>>> positives = filter(lambda x: x > 0, [1, -5, 0, 6, -2, 8])
>>> positives
<filter object at 0x10fe9d490>
>>> list(positives)
[1, 6, 8]
>>>
```

Passing `None` as the first argument to `filter()` will filter out input elements which evaluate to `False`.

Filtering with None

```
>>> trues = filter(None, [0, 1, False, True, [], [1, 2, 3], '', 'hello'])
>>> list(trues)
[1, True, [1, 2, 3], 'hello']
>>>
```

Python 2 vs. Python 3

`map()` and `filter()` behave differently in Python 2 and Python 3

In Python 3, they are lazy

In Python2, they are eager and return lists

functools.reduce()

`functools.reduce()`

Repeatedly applies a two-argument function to an accumulated value and the next element from a sequence

The initial value can be the first element in the input sequence or an optional argument

The final accumulated - or reduced - value is returned `reduce()` is not unique to Python

`fold()` in many functional languages

`Aggregate()` in .NET's LINQ

`accumulate()` in C++'s Standard Template Library

reduce()

```
mul 120 6
mul 720 7
mul 5040 8
mul 40320 9
362880
>>> reduce(mul, [])
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: reduce() of empty sequence with no initial value
>>> reduce(mul, [1])
1
>>> values = [1, 2, 3]
>>> reduce(operator.add, values, 0)
6
>>> values = []
>>> reduce(operator.add, values, 0)
0
>>> values = [1, 2, 3]
>>> reduce(operator.add, values, 0)
6
>>> values = [1, 2, 3]
>>> reduce(operator.mul, values, 1)
6
>>>
```

`reduce()` accepts an optional initial value.

Conceptually added to the start of the sequence.

Serves as the first accumulator value.

Map-reduce

Are `map()` and `reduce()` related to map-reduce?

Yes! They are the core concepts in that algorithm.

Map-reduce

```
...     'It was the best of times, it was the worst of times.',
...     'I went to the woods because I wished to live deliberately, to front onl
y the essential facts of life...',
...     'Friends, Romans, countrymen, lend me your ears; I come to bury Caesar,
not to praise him.',
...     'I do not like green eggs and ham. I do not like them, Sam-I-Am.',
... ]
>>> counts = map(count_words, documents)
>>> def combine_counts(d1, d2):
...     d = d1.copy()
...     for word, count in d2.items():
...         d[word] = d.get(word, 0) + count
...     return d
...
>>> from functools import reduce
>>> total_counts = reduce(combine_counts, counts)
>>> total_counts
{'it': 2, 'was': 2, 'the': 4, 'best': 1, 'of': 3, 'times': 2, 'worst': 1, 'i': 6
, 'went': 1, 'to': 5, 'woods': 1, 'because': 1, 'wished': 1, 'live': 1, 'deliber
ately': 1, 'front': 1, 'only': 1, 'essential': 1, 'facts': 1, 'life': 1, 'friend
s': 1, 'romans': 1, 'countrymen': 1, 'lend': 1, 'me': 1, 'your': 1, 'ears': 1, '
come': 1, 'bury': 1, 'caesar': 1, 'not': 3, 'praise': 1, 'him': 1, 'do': 2, 'lik
e': 2, 'green': 1, 'eggs': 1, 'and': 1, 'ham': 1, 'them': 1, 'sam': 1, 'am': 1}
>>>
```

Summary

`map()` applies a callable to each element in a sequence

`map()` produces its results lazily

`map()` can accept multiple input iterables

`filter()` applies a predicate to the elements of an iterable

It produces an iterable containing the input elements for which the predicate returned `True`

Summary

`functools.reduce()`

- Repeatedly applies a two-argument callable to accumulate the elements in an iterable
- Raises an exception on empty input iterables
- You can provide an initial value to avoid this issue
- Selecting the right initial value is crucial

Combining `map()` and `reduce()` to make map-reduce