Implementing Collections

Robert Smallshire
@robsmallshire
rob@sixty-north.com

Presenter

Austin Bingham
@austin_bingham
austin@sixty-north.com

pluralsight
hardcore dev and IT training

Collections

```
tuple
str
range
list
dict
set
```

Collection Protocols

Protocol	Implementing Collections
Container	`str, list, range, tuple, set, bytes, dict`
Sized	`str, list, range, tuple, set, bytes, dict`
Iterable	`str, list, range, tuple, set, bytes, dict`
Sequence	`str, list, range, tuple, bytes`
Set	`set`
Mutable Sequence	`list`
Mutable Set	`set`
Mutable Mapping	`dict`

Collection Protocols

Protocol	Implementing Collections
Container	`str, list, range, tuple, set, bytes, dict`
Sized	`str, list, range, tuple, set, bytes, dict`
Iterable	`str, list, range, tuple, set, bytes, dict`
Sequence	`str, list, range, tuple, bytes`
Set	`set`
Mutable Sequence	`list`
Mutable Set	`set`
Mutable Mapping	`dict`

Protocols

- To implement a protocol, objects must support certain operations.
- Most collections implement container, sized and iterable.
- All except `dict` and `set` are sequences.

Collection Protocols

Protocol	Implementing Collections
Container	`str, list, range, tuple, set, bytes, dict`
Sized	`str, list, range`
Iterable	
Sequence	
Set	`set`
Mutable Sequence	`list`
Mutable Set	`set`
Mutable Mapping	`dict`

Container Protocol

- Membership testing using `in` and `not in`

Collection Protocols

Protocol	Implementing Collections
Container	`str, list, range, tuple, set, bytes, dict`
Sized	`str, list, range, tuple, set, bytes, dict`
Iterable	`str, list, range, tuple, set, bytes, dict`
Sequence	`str`
Set	`se`
Mutable Sequence	`list`
Mutable Set	`set`
Mutable Mapping	`dict`

Sized Protocol

- Determine number of elements with `len(s)`

Collection Protocols

Protocol	Implementing Collections
Container	`str, list, range, tuple, set, bytes, dict`
Sized	`str, list, range, tuple, set, bytes, dict`
Iterable	`str, list, range, tuple, set, bytes, dict`
Sequence	`str, list, range, t`
Set	`set`
Mutable Sequence	`list`
Mutable Set	`set`
Mutable Mapping	`dict`

Iterable Protocol

- Can produce an *iterator* with `iter(s)`

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

Collection Protocols

Protocol	Implementing Collections
Container	`str, list, range, tuple, set, bytes, dict`
Sized	`str, list, range, tuple, set, bytes, dict`
Iterable	`str, list, range, tuple, set, bytes, dict`
Sequence	`str, list, range, tuple, bytes`
Set	`set`
Mutable Sequence	`list`
Mutable Set	`set`
Mutable Mapping	`dict`

Sequence Protocol

- Retrieve elements by index

```
item = seq[index]
```

- Find items by value

```
index = seq.index(item)
```

- Count items

```
num = seq.count(item)
```

- Produce a reversed sequence

```
r = reversed(seq)
```

Collection Protocols

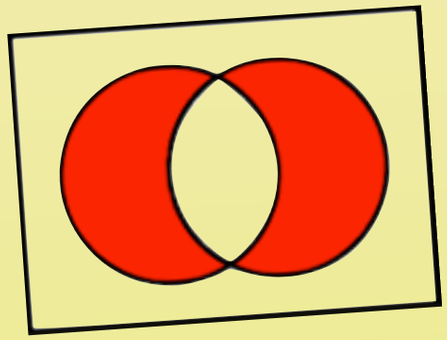
Protocol	Implementing Collections
Container	`str, list, range, tu`
Sized	`str, list, range, tup`
Iterable	`str, list, range, tup`
Sequence	`str, list, range, tup`
Set	`set`
Mutable Sequence	`list`
Mutable Set	`set`
Mutable Mapping	`dict`

Set Protocol

set algebra operations

(**methods** and infix **operators**)

- subset
- proper subset
- equal
- not equal
- proper superset
- superset
- intersections
- union
- symmetric difference
- difference

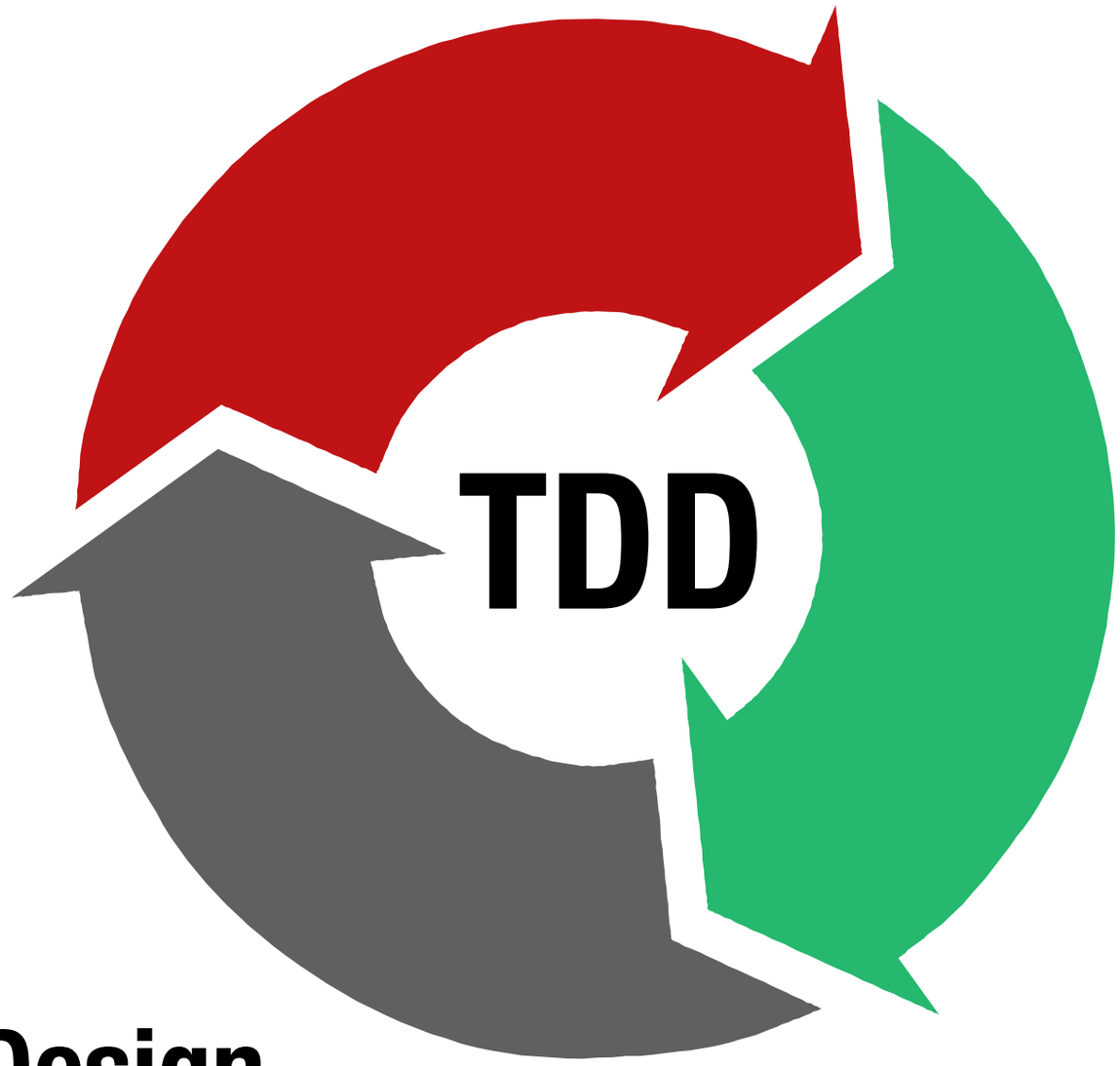


Let's build a

SortedSet

A collection which is a
sized, iterable, sequence
container of a set of
distinct items and
constructible from an
iterable

Test



Code

Design
(Refactor)

The construction convention

```
collection_from_iterable = Collection(iterable)

empty_collection = Collection()
```

The container protocol

- Membership testing using `in` and `not in`
- Special method: `__contains__(item)`
- Fallback to **iterable** protocol

The sized protocol

- Number of items using `len(sized)` function
- Must **not** consume or modify collection
- Special method: `__len__()`

The iterable protocol

- Obtain an iterator with `iter(iterable)` function
- Special method: `__iter__()`

The sequence protocol

Implies **container**, **sized** and **iterable**

- Retrieve slices by slicing

```
item = seq[index]
```

- Retrieve slices by slicing

```
item = seq[start:stop]
```

- Special method `__getitem__()`

- Produce a reversed sequence

```
r = reversed(seq)
```

- Special method `__reversed__()`
- Fallback to `__getitem__()` and `__len__()`

- Find items by value

```
index = seq.index(item)
```

- No special method

- Count items

```
num = seq.count(item)
```

- No special method

- Concatenation with `+` operator
- Special method `__add__()`

- Repetition with `*` operator
- Special methods `__mul__()` and `__rmul__()`

equality and inequality

- Equality

  `lhs == rhs`

- Special method

  `__eq__(self, rhs)`

- `self` argument is `lhs` left-hand-side operand

- Inequality

  `lhs != rhs`

- Special method

  `__ne__(self, rhs)`

- `self` argument is `lhs` left-hand-side operand

`object.__lt__(self, other)`
`object.__le__(self, other)` ¶
`object.__eq__(self, other)`
`object.__ne__(self, other)`
`object.__gt__(self, other)`
`object.__ge__(self, other)`

These are the so-called "rich comparison" methods. The correspondence between operator symbols and method names is as follows: `x<y` calls `x.__lt__(y)`, `x<=y` calls `x.__le__(y)`, `x==y` calls `x.__eq__(y)`, `x!=y` calls `x.__ne__(y)`, `x>y` calls `x.__gt__(y)`, and `x>=y` calls `x.__ge__(y)`.

A rich comparison method may return the singleton `NotImplemented` if it does not implement the operation for a given pair of arguments. By convention, `False` and `True` are returned for a successful comparison. However, these methods can return any value, so if the comparison operator is used in a Boolean context (e.g., in the condition of an `if` statement), Python will call `bool()` on the value to determine if the result is true or false.

There are no implied relationships among the comparison operators. The truth of `x==y` does not imply that `x!=y` is false. Accordingly, when defining `__eq__()`, one should also define `__ne__()` so that the operators will behave as expected. See the paragraph on `__hash__()` for some important notes on creating *hashable* objects which support custom comparison operations and are usable as dictionary keys.

There are no swapped-argument versions of these methods (to be used when the left argument does not support the operation but the right argument does); rather, `__lt__()` and `__gt__()` are each other's reflection, `__le__()` and `__ge__()` are each other's reflection, and `__eq__()` and `__ne__()` are their own reflection.

Arguments to rich comparison methods are never coerced.

To automatically generate ordering operations from a single root operation, see

- Produce a reversed sequence

```
r = reversed(seq)
```

- Special method `__reversed__()`
- Fallback to `__getitem__()` and `__len__()`

8.4. collections.abc — Abstract Base Classes for Containers

New in version 3.3: Formerly, this module was part of the `collections` module.

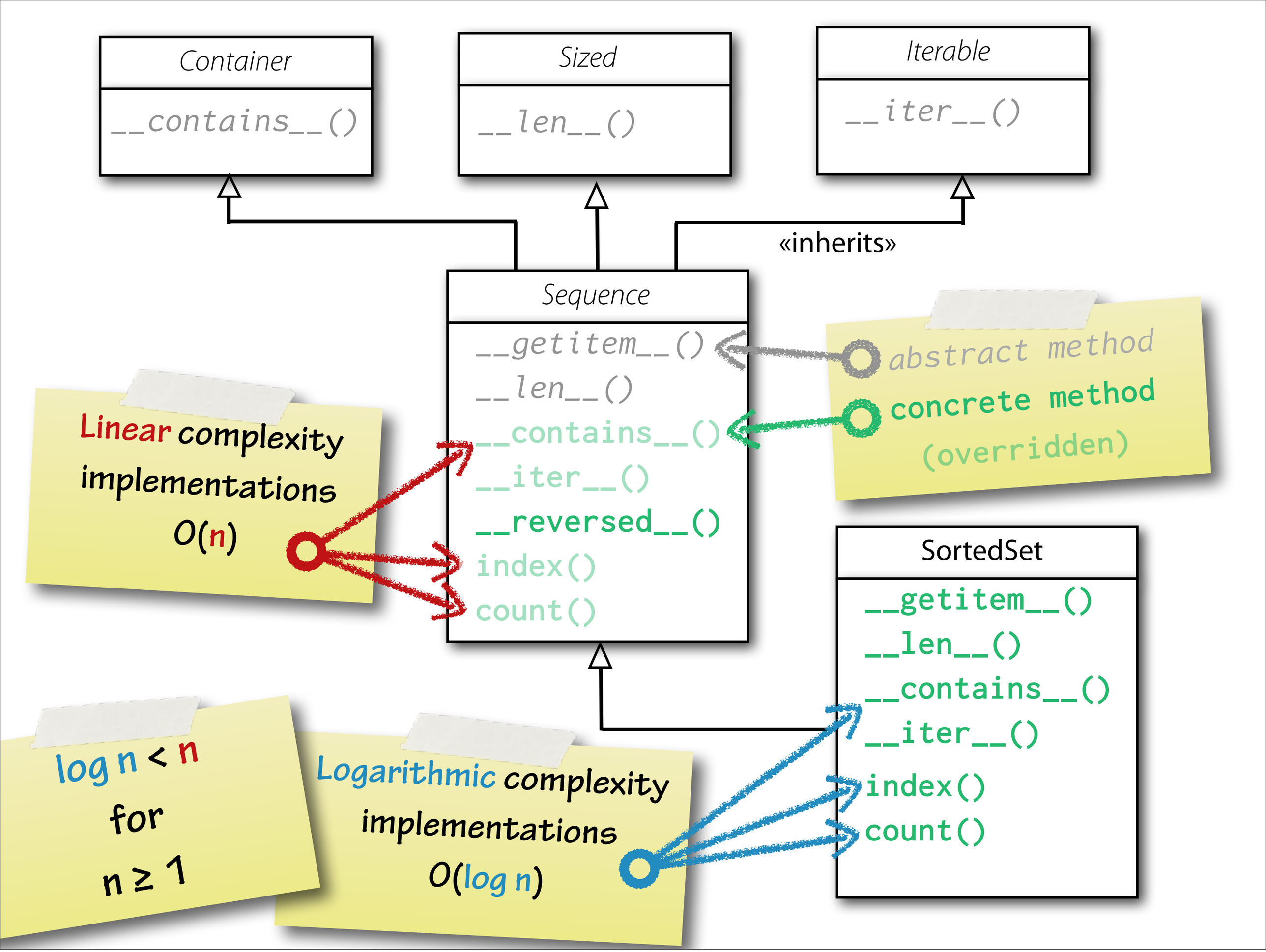
Source code: Lib/_collections_abc.py

This module provides *abstract base classes* that can be used to test whether a class provides a particular interface; for example, whether it is hashable or whether it is a mapping.

8.4.1. Collections Abstract Base Classes

The collections module offers the following *ABCs*:

ABC	Inherits from	Abstract Methods	Mixin Methods
Container		`__contains__`	
Hashable		`__hash__`	
Iterable		`__iter__`	
Iterator	Iterable	`__next__`	`__iter__`
Sized		`__len__`	
Callable		`__call__`	
Sequence	Sized, Iterable, Container	`__getitem__`, `__len__`	`__contains__`, `__iter__`, `__reversed__`, `index`, and `count`
MutableSequence	Sequence	`__getitem__`,	Inherited Sequence methods and


Container
__contains__()
Sized
__len__()
Iterable
__iter__()
«inherits»
Sequence
__getitem__()
__len__()
__contains__()
__iter__()
__reversed__()
index()
count()
abstract method
concrete method
(overridden)
Linear complexity
implementations
O(n)
log n < n
for
n ≥ 1
Logarithmic complexity
implementations
O(log n)
SortedSet
__getitem__()
__len__()
__contains__()
__iter__()
index()
count()

ABCs vs. Duck Typing

Does the introduction of ABCs mean the end of Duck Typing? I don't think so. Python will not require that a class derives from `BasicMapping` or `Sequence` when it defines a `__getitem__` method, nor will the `x[y]` syntax require that `x` is an instance of either ABC. You will still be able to assign any "file-like" object to `sys.stdout`, as long as it has a `write` method.

Of course, there will be some carrots to encourage users to derive from the appropriate base classes; these vary from default implementations for certain functionality to an improved ability to distinguish between mappings and sequences. But there are no sticks. If `hasattr(x, "__len__")` works for you, great! ABCs are intended to solve problems that don't have a good solution at all in Python 2, such as distinguishing between mappings and sequences.

ABCs vs. Generic Functions

8.4. `collections.abc` — Abstract Base Classes for Containers

New in version 3.3: Formerly, this module was part of the `collections` module.

Source code: Lib/collections/abc.py

This module provides *abstract base classes* that can be used to test whether a class provides a particular interface; for example, whether it is hashable or whether it is a mapping.

8.4.1. Collections Abstract Base Classes

The collections module offers the following *ABCs*:

ABC	Inherits from	Abstract Methods	Mixin Methods
`Container`		`__contains__`	
`Hashable`		`__hash__`	
`Iterable`		`__iter__`	
`Iterator`	`Iterable`	`__next__`	`__iter__`
`Sized`		`__len__`	
`Callable`		`__call__`	
`Sequence`	`Sized`, `Iterable`, `Container`	`__getitem__`, `__len__`	`__contains__`, `__iter__`, `__reversed__`, `index`, and `count`
`MutableSequence`	`Sequence`	`__getitem__`, `__setitem__`, `__delitem__`, `__len__`, `insert`	Inherited `Sequence` methods and `append`, `reverse`, `extend`, `pop`, `remove`, and `__iadd__`
`Set`	`Sized`, `Iterable`, `Container`	`__contains__`, `__iter__`, `__len__`	`__le__`, `__lt__`, `__eq__`, `__ne__`, `__gt__`, `__ge__`, `__and__`, `__or__`, `__sub__`, `__xor__`, and `isdisjoint`
`MutableSet`	`Set`	`__contains__`, `__iter__`, `__len__`, `add`, `discard`	Inherited `Set` methods and `clear`, `pop`, `remove`, `__ior__`, `__iand__`, `__ixor__`, and `__isub__`
`Mapping`	`Sized`, `Iterable`, `Container`	`__getitem__`, `__iter__`, `__len__`	`__contains__`, `keys`, `items`, `values`, `get`, `__eq__`, and `__ne__`
`MutableMapping`	`Mapping`	`__getitem__`, `__setitem__`,	Inherited `Mapping` methods and `pop`, `popitem`, `clear`, `update`, and `setdefault`

The set protocol

Implies **container**, **sized** and **iterable**

Relational operators

special method	infix operator	set method	meaning
`__le__()`	`<=`	`issubset()`	subset
`__lt__()`	`<`		proper subset
`__eq__()`	`==`		equal
`__ne__()`	`!=`		not equal
`__gt__()`	`>`		proper superset
`__ge__()`	`>=`	`issuperset()`	superset
		`isdisjoint()`	disjoint

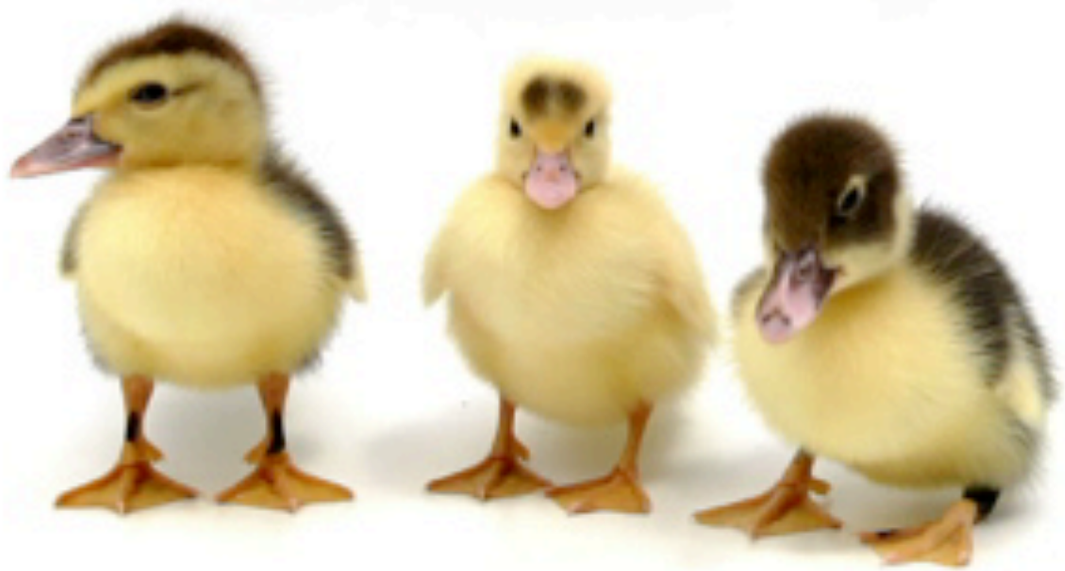
As implemented by built-in `set`

Provided by `collections.abc.Set`

Algebraic operators

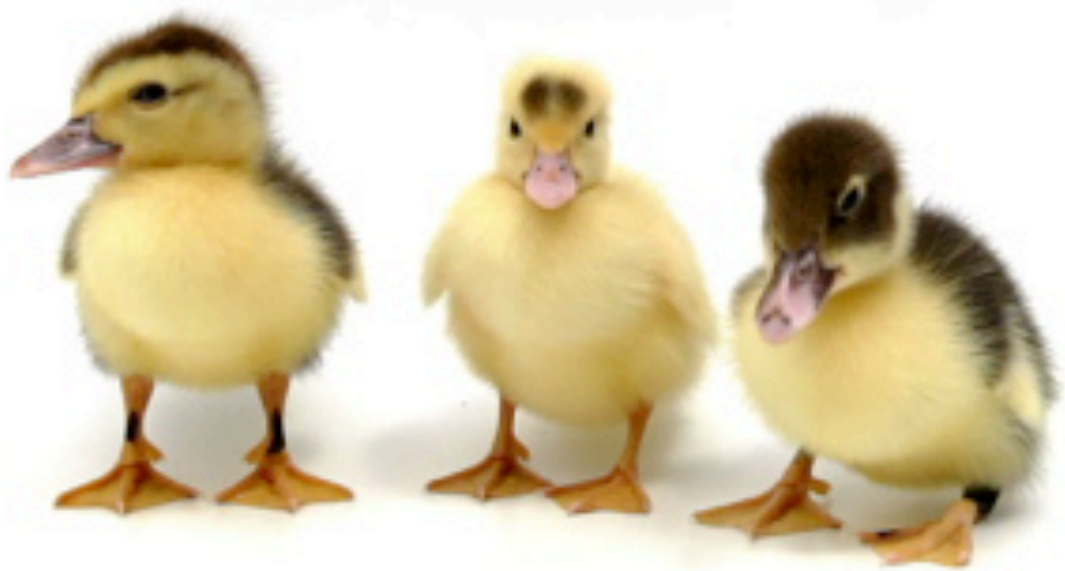
special method	infix operator	set method
`__and__()`	`&`	`intersection()`
`__or__()`	`|`	`union()`
`__xor__()`	`^`	`symmetric_difference()`
`__sub__()`	`-`	`difference()`

Duck Tails



Tales of real-world Python
– better in practice than in theory –
just like Duck Typing.

Duck Tails



Duck Tails

Mutable Sets
An Exercise for the Viewer

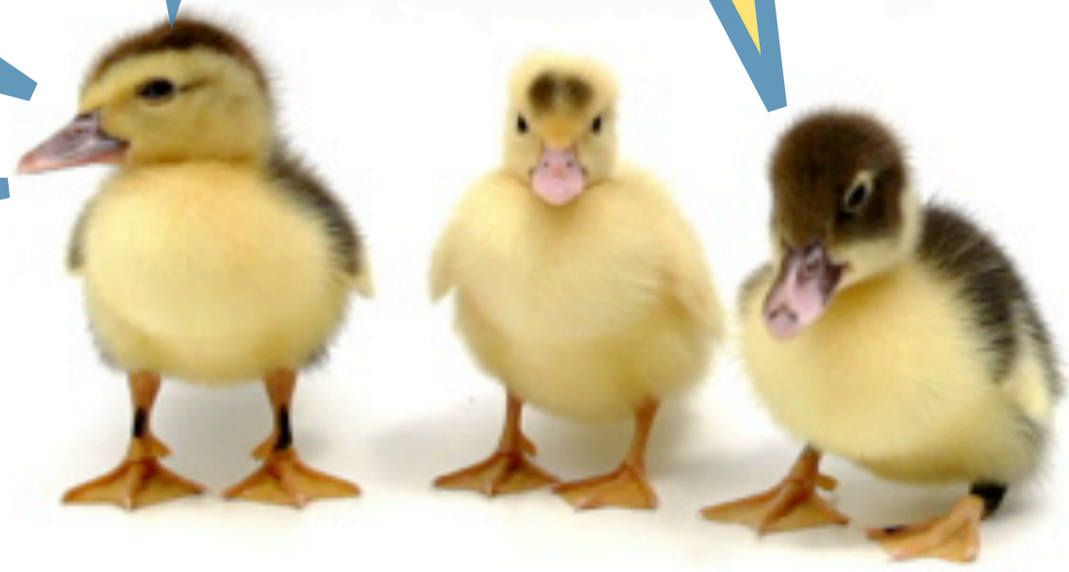
Duck Tails

CONSIDER ADDING `update()` AND `symmetric_difference_update()`, ETC.

IMPLEMENT `add()` AND `discard()`

REVISIT ASSUMPTIONS!

INHERIT FROM MUTABLESET INSTEAD OF SET



Implementing Collections

collection protocols

`collections.abc`

Container	Sized	Iterable	Sequence	Set
container protocol	**sized protocol**	**iterable protocol**	**sequence protocol**	**set protocol**

`__contains__()`	`__len__()`	`__iter__()`	`__getitem__()`
			`__reversed__()`
			`index()`
			`count()`
			`__add__()`
			`__mul__()`
			`__rmul__()`
				`__le__()`
				`__lt__()`
				`__eq__()`
				`__ne__()`
				`__gt__()`
				`__ge__()`
				`__and__()`
				`__or__()`
				`__xor__()`
				`__sub__()`
				`isdisjoint()`

string representation	**value equality**			
`__repr__()`	`__eq__()` `__ne__()`			