
openrange Documentation

Release 0.0.1

Josh Tomlinson

March 20, 2015

1	BaseRange	1
1.1	Example	1
2	Range	3
2.1	enumeration	3
2.2	exclusion	3
2.3	random iteration	4
2.4	repeat iteration	4
3	datetime Ranges	5
3.1	DateRange	5
3.2	DatetimeRange	5
3.3	TimeRange	5
4	API	7
4.1	BaseRange	7
4.2	Range	8
4.3	datetime Ranges	8
5	OpenRange	9
5.1	Overview	9
5.2	Indices and tables	10
	Python Module Index	11

BaseRange

BaseRange is an abstract base class that provides the common interface for all **OpenRange** objects. Like the built-in range object, BaseRange is a subclass of `Sequence` and supports all of the common sequence operations. The constructor for BaseRange uses the same arguments and defaults as the built-in `range`.

Subclasses of BaseRange need only define how to convert between the type of objects within the progression and an underlying numeric type. To do so, these two abstract methods must be implemented:

```
@abstractmethod
def _item_to_num(self, item):
    """Convert the item to a numerical value."""

@abstractmethod
def _num_to_item(self, num):
    """Convert the value to an item in the progression."""
```

For example, to implement a range-like object that generates `datetime.date` objects, `_item_to_num` would convert a `datetime.date` item to a numerical representation like seconds since the epoch. Conversely, `_num_to_item` would convert seconds since the epoch back to a `datetime.date` object.

Once these two methods are implemented, everything else is handled by BaseRange.

In some cases, the `step` type may differ from the items within the progression. In this case, a subclass should implement the following conversion methods:

```
@abstractmethod
def _step_to_num(self, step):
    """Convert supplied step item to a numeric value."""

@abstractmethod
def _num_to_step(self, num):
    """Convert supplied numeric value to a step item."""
```

For the `datetime.date` example, the `step` would be implemented as a `datetime.timedelta` object. The `_step_to_num` method would convert a `datetime.timedelta` object to seconds whereas `_num_to_step` would convert seconds back to a `datetime.timedelta` object.

The default implementations of the `step` conversion methods assume the `step` is of the same type as `start` and `stop`, and therefore fall back to calling the `_item_to_num` and `_num_to_item` methods.

1.1 Example

Here's a simple, yet full implementation of a range-like object that iterates over strings representing binary numbers.

```
from openrange import BaseRange

class BinaryStrRange(BaseRange):

    def _item_to_num(self, item):
        return int(str(item), 2)

    def _num_to_item(self, num):
        return "{n:b}".format(n=num)

for i in BinaryStrRange("1000"):
    print i,

# prints:
# 0 1 10 11 100 101 110 111 1000
```

You can see how the two required methods, `_item_to_num` and `_num_to_item` convert between string and integer values. You can also see the default value for `start` is 0 and `step` is 1, just like the built-in `range`.

Note: You may have noticed that the `BaseRange` implementation is not quite identical to the built-in `range`. Unlike the built-in `range`, `BaseRange` implements iteration as inclusive of the `stop` value. The built-in `range` is exclusive of the `stop` value because it is commonly used to generate integers for zero-based indexing of lists. The typical usage of `BaseRange` will likely not be to generate integer types and so the decision was made to make the iteration inclusive of the `stop` value.

Note: Like python 2's built-in `xrange` and python 3's built-in `range` object, `BaseRange` does its best to avoid evaluating items in the progression until it has to. In cases where this is unavoidable, that method's documentation will say so.

Range

OpenRange comes with a generic numerical range-like class called `Range`. This class inherits `BaseRange` and supports any numeric type (`float`, `int`, `decimal.Decimal`, etc.) for its `start`, `stop`, and `step` values. Iterating over a `Range` object yields `int` and/or `float` items depending on the values within the progression.

The primary purpose of `Range` is for testing `BaseRange`, but it can also be used to show some of the additional features that `BaseRange` provides that don't exist in the built-in `range`. These features are highlighted in the sections below.

2.1 enumeration

An `enumerate` method is available for generating tuples of the form `(count, item)` for items within the progression. The method is similar to python's built-in `enumerate` method, including the optional `start` argument.

```
>>> from openrange.rng import Range
>>> for i in Range(-1.0, 1, .5).enumerate():
...     print str(i),
...
(0, -1) (1, -0.5) (2, 0.0) (3, 0.5) (4, 1.0)

>>> for i in Range(-1, 1, .5).enumerate(start=5):
...     print str(i),
...
(5, -1) (6, -0.5) (7, 0.0) (8, 0.5) (9, 1.0)
```

2.2 exclusion

`BaseRange` subclasses allow iteration over a progression with the ability to exclude certain items. This is possible using the `excluding` method supplied with a list of items to exclude. The items in the iterable should be of the same type as the object's `start` and `stop` arguments.

```
>>> from openrange.rng import Range
>>> for i in Range(-1.0, 1, .5).excluding([0, 1, 10]):
...     print str(i),
...
-1 -0.5 0.5
```

2.3 random iteration

Another feature of `BaseRange` subclasses is the ability to iterate over items in the progression in a random order using the `random` method.

```
>>> from openrange.rng import Range
>>> for i in Range(-1.0, 1, .5).random():
...     print str(i),
...
1.0 0.5 -1.0 -0.5 0.0
```

2.4 repeat iteration

For cases where iterating of the progression multiple times is useful, the `repeat` method can be used. By default, it will generate the items in the progression 2 times. The optional `times` argument can be used to repeat the items more than twice.

```
>>> from openrange.rng import Range
>>> for i in Range(-1, 1, .5).repeat():
...     print str(i),
...
-1 -0.5 0.0 0.5 1.0 -1 -0.5 0.0 0.5 1.0

>>> for i in Range(-1, 1, .5).repeat(times=3):
...     print str(i),
...
-1 -0.5 0.0 0.5 1.0 -1 -0.5 0.0 0.5 1.0 -1 -0.5 0.0 0.5 1.0
```

datetime Ranges

OpenRange comes with 3 additional example range-like implementations based on types defined in python's `datetime` module. These objects are highlighted in the following sections.

3.1 DateRange

`DateRange` generates `datetime.date` objects between given start and stop `datetime.date` objects. The step value is provided as a `datetime.timedelta` object. Here are some examples:

```
# coming soon...
```

3.2 DatetimeRange

`DatetimeRange` generates `datetime.datetime` objects between given start and stop `datetime.datetime` objects. The step value is provided as a `datetime.timedelta` object. Here are some examples:

```
# coming soon...
```

3.3 TimeRange

`TimeRange` generates `datetime.time` objects between given start and stop `datetime.time` objects. The step value is provided as a `datetime.timedelta` object. Here are some examples:

```
# coming soon...
```


4.1 BaseRange

Create custom arithmetic progression classes.

```
class openrange.base.BaseRange (*args)
    Bases: _abcoll.Sequence
```

Abstract base class for custom arithmetic progressions.

Subclasses need only define how to convert between the type of objects within the progression and an underlying numeric type. To do so, these abstract methods must be implemented:

```
_item_to_num(self, item) _num_to_item(self, num)
```

In some cases, the step type may differ from the items within the progression. In this case, a subclass should implement the following conversion methods:

```
_step_to_num(self, step) _num_to_step(self, num)
```

The default implementations of these step conversion methods assume the start, stop, and step are of the same type and therefore call the abstract `_item_to_num()` and `_num_to_item()` methods.

count (*item*)

Returns the number of times item appears in the progression.

enumerate (*start=0*)

Generates tuples for each item in the progression.

The tuples yielded take the form (count, item). Count starts at 0 unless an optional keyword argument 'start' is supplied with an alternate start value.

excluding (*iterable*)

Iterate over progression excluding items in supplied iterable.

index (*item*)

Returns the index of the first item matching the supplied item.

random ()

Generate the items in the progression in a random order.

repeat (*times=2*)

Iterate over the progression multiple times in sequence.

reverse ()

Reverses the range in place.

start

The start item for this range.

step

The step item for this range.

stop

The stop item for this range.

4.2 Range

class `openrange.rng.Range(*args)`

Bases: `openrange.base.BaseRange`

Inclusive numerical range.

4.3 datetime Ranges

class `openrange.dt.DateRange(start, stop, step)`

Bases: `openrange.base.BaseRange`

Date object progression.

class `openrange.dt.DatetimeRange(start, stop, step)`

Bases: `openrange.base.BaseRange`

Datetime object progression.

class `openrange.dt.TimeRange(start, stop, step)`

Bases: `openrange.base.BaseRange`

Time object progression.

OpenRange

OpenRange provides a simple interface for building custom range-like objects for any type that can be represented numerically.

5.1 Overview

Python's built-in `range` is great for generating a list of integers and when iterating over the indices of a sequence. There are times, however, when you'd like a similar interface for non-integer types.

The idea behind **OpenRange** is to provide a base class that allows for quick implementation of arithmetic progressions for any type that can be represented numerically. For example, you might be interested in a range-like interface for iterating over a `datetime.date` objects using `datetime.timedelta` as the step. **OpenRange** provides an example implementation that does just that:

```
import datetime
from openrange.dt import DateRange

start_date = datetime.date.today()
end_date = start_date + datetime.timedelta(days=365)
two_weeks = datetime.timedelta(days=14)

# yield datetime.date objects for every 2 weeks, starting today, for a year
for dt_date in DateRange(start_date, end_date, two_weeks):
    # ... profit
```

OpenRange makes implementing these types of classes very simple by providing an easy-to-use abstract base class called `BaseRange`. See the full Documentation for more info.

5.1.1 Installation

OpenRange is easy to install using pip.

```
$ pip install openrange
```

5.1.2 Support

OpenRange is tested against:

- python 2.7, 3.2, 3.3, 3.4

- `pypy` and `pypy3`

Primary development and testing were for python 2.7.

5.1.3 Contribute

Contribution is welcome from those who propose new features, have ideas for improvement, or submit bug fixes. Here's a checklist for contributing to this project:

- Open or respond to an issue to discuss a feature or bug
- Fork the repo on GitHub and start making changes
- Write test(s) for the bug or feature
- Add yourself to **CONTRIBUTORS.rst**
- Send a pull request

5.2 Indices and tables

- *genindex*
- *modindex*
- *search*

O

`openrange.base`, 7

`openrange.dt`, 8

`openrange.rng`, 8

B

BaseRange (class in openrange.base), 7

C

count() (openrange.base.BaseRange method), 7

D

DateRange (class in openrange.dt), 8

DatetimeRange (class in openrange.dt), 8

E

enumerate() (openrange.base.BaseRange method), 7

excluding() (openrange.base.BaseRange method), 7

I

index() (openrange.base.BaseRange method), 7

O

openrange.base (module), 7

openrange.dt (module), 8

openrange.rng (module), 8

R

random() (openrange.base.BaseRange method), 7

Range (class in openrange.rng), 8

repeat() (openrange.base.BaseRange method), 7

reverse() (openrange.base.BaseRange method), 7

S

start (openrange.base.BaseRange attribute), 7

step (openrange.base.BaseRange attribute), 8

stop (openrange.base.BaseRange attribute), 8

T

TimeRange (class in openrange.dt), 8