types-linq

cleoold

Jun 24, 2023

CONTENTS

`types-linq` is a lightweight Python library that attempts to implement LINQ (Language Integrated Query) features seen in .NET languages (see here for their documentation).

This library provides similarly expressive and unified querying exprience on objects so long as it is iterable. With few simple method calls and lambdas, developers can perform complex traversal, filter and transformations on any data that typically had to be done with many iterative logics such as for loops.

There have been several libraries that try providing such functionalities, while this library tries to accomplish something different:

- It incorporates the original APIs in .NET `IEnumerable` class as close as possible, including method names, conventions, edge behaviors, etc. This means typical Python conventions might be shadowed here
- It tries to implement deferred evaluations. The library operates in a streaming manner if possible and handles infinite streams (Python generators) properly
- Strong type safety while using this library is guarenteed since the APIs are typed
- It honours the Python collections.abc interfaces

The project is licensed under the BSD-2-Clause License.

CHAPTER

ONE

INSTALLING

1.1 From pypi

The project is available on pypi, to install the letest version, do:

```
$ pip install types-linq -U
```

1.2 From GitHub Repo

Clone the project and install from local files:

```
$ git clone https://github.com/cleoold/types-linq && cd types-linq
$ pip install .
# or
$ python setup.py install
```

1.2.1 Run Tests

In the project root, execute the following commands (or something similar) to run the test cases:

```
# optionally set up venv
$ python -m venv
$ ./scripts/activate

$ pip install pytest
$ python -m pytest
```

If you want to run the project against pyright, the following should do:

```
$ npm install pyright -g
$ npx pyright
```

Instead, opening vscode should also highlight red striggles (?)

However, the GitHub action settings are most up-to-date and can be consulted.

1.2.2 Build the Documentation

To generate the pages you are currently looking at, in the project root, execute the following commands:

```
$ cd doc
$ pip install -r requirements.txt
# generate api md files
$ python ./gen_api_doc.py
# create html pages, contents are available in _build/html folder
$ make html
```

Note to generate api files, one must have Python version 3.9 or above. The api md files are committed to the repository.

CHAPTER

TWO

EXAMPLES

The primary class importable from this library is the `Enumerable` class. To query on an existing object such as lists, tuples, or generators, you pass the object to the `Enumerable` constructor, then invoke chained methods like this:

```
from types_linq import Enumerable

lst = [1, 4, 7, 9, 16]

query = Enumerable(lst).where(lambda x: x % 2 == 0).select(lambda x: x ** 2)

for x in query:
    print(x)
```

This will filter the list by whether the element is even, then converts each element to the square of it. The call to `where` and `select` will return immediately. Finally when the iterator of `query` is requested in the for loop, the element will be enumerated in the order.

It is roughly equivalent to the following code:

```
for x in lst:
    if x % 2 == 0:
        print(x ** 2)
```

or

```
for x in map(lambda x: x ** 2, filter(lambda x: x % 2 == 0, lst)):
    print(x)
```

The output will be `16` and `256` printed on newlines.

The class supports a lot more than this. The usage is simple if you know about the interfaces in .NET as this library provides almost the exact methods. It is advised to take a look at the tests to digest more in-action use cases.

2.1 More examples

Grouping and transforming lists:

```
from typing import NamedTuple
from types_linq import Enumerable as En


```

(continues on next page)

(continued from previous page)

```
class AnswerSheet(NamedTuple):
    subject: str
    score: int
    name: str

students = ['Jacque', 'Franklin', 'Romeo']
papers = [
    AnswerSheet(subject='Calculus', score=78, name='Jacque'),
    AnswerSheet(subject='Calculus', score=98, name='Romeo'),
    AnswerSheet(subject='Algorithms', score=59, name='Romeo'),
    AnswerSheet(subject='Mechanics', score=93, name='Jacque'),
    AnswerSheet(subject='E & M', score=87, name='Jacque'),
]

query = En(students) \
    .order_by(lambda student: student) \
    .group_join(papers,
        lambda student: student,
        lambda paper: paper.name,
        lambda student, papers: {
            'student': student,
            'papers': papers.order_by(lambda paper: paper.subject) \
                .select(lambda paper: {
                    'subject': paper.subject,
                    'score': paper.score,
                }).to_list(),
            'gpa': papers.average2(lambda paper: paper.score, None),
        }
    )

for obj in query:
    print(obj)

# output:
# {'student': 'Franklin', 'papers': [], 'gpa': None}
# {'student': 'Jacque', 'papers': [{'subject': 'E & M', 'score': 87}, {'subject': 'Mechanics',
↪'score': 93}, {'subject': 'Calculus', 'score': 78}], 'gpa': 86.0}
# {'student': 'Romeo', 'papers': [{'subject': 'Algorithms', 'score': 59}, {'subject': 'Calculus',
↪'score': 98}], 'gpa': 78.5}
```

Working with generators:

```
import random
from types_linq import Enumerable as En

def toss_coins():
    while True:
        yield random.choice(('Head', 'Tail'))

times_head = En(toss_coins()).take(5) \  # [:5] also works
    .count(lambda r: r == 'Head')
```

(continues on next page)

(continued from previous page)

```
print(f'You tossed 5 times with {times_head} HEADs!')

# possible output:
# You tossed 5 times with 2 HEADs!
```

Working with stream output:

```
import sys, subprocess
from types_linq import Enumerable as En

proc = subprocess.Popen('kubectl logs -f my-pod', shell=True, stdout=subprocess.PIPE)
stdout = iter(proc.stdout.readline, b'')

query = En(stdout).where(lambda line: line.startswith(b'CRITICAL: ')) \
    .select(lambda line: line[10:].decode())

for line in query:
    sys.stdout.write(line)
    sys.stdout.flush()

# whatever.
```

CHAPTER

THREE

DIFFERENCES FROM *"IENUMERABLE<OUT T>"*

Because Python and C# are two languages that have a lot of differences, this library does not intimate everything from the .NET world as some practices are not possible Python world. This section lists some differences (or limitations) between the `types_linq.Enumerable` class and its .NET counterpart.

- In C#, there are extension methods. By `using` the correct namespaces, the query methods will be automatically available on all references to `IEnumerable` variables. Such concepts do not exist in Python, hence users have to wrap the object under a `types_linq.Enumerable` class to use those query methods.
- C# uses overloading extensively while there are no real method overloading in Python. Rather, to define overload methods in Python, one must use the `typing.overload` decorator to decorate stubs, then implement all overloads together in a single definition. An example can be found here. The downside of this is that the features supported are quite limited.

  For example, it can be simple to seperate between `def fn(a: int) -> None` and `def fn(a: int, b: int) -> None`, also between `def fn(a: str) -> None` and `def fn(a: bytes) -> None` by checking the number of arguments or using `isinstance()`. However, when it comes to separate `def fn(a: Callable[[TSource_co], bool]) -> None` and `def fn(Callable[[TSource_co, int], bool]) -> None`, there is no straightforward way that works for all occasions (typical reflection check will fail for C extensions, and try-except is impractical). This library's solution is a sketchy one: using different names for these methods, for example, `Enumerable.where()` and `Enumerable.where2()`. This is the reason why some names of methods here end with numbers.

  It can also bring some troubles when disambiguating types that overlap. If an object implements both `Iterable` and `Callable`, and there are method overloads for each, the behavior might be inconsistent if the implementation does not agree with stubs. Type checkers will pick the first matching overload.
- There are no "IEqualityComparer" or something like that in Python. C# people will use these to compare objects, construct hashmaps, etc. While in Python such identities are often solely determined by object's magic methods such as `__hash__()`, `__eq__()`, `__lt__()`, etc. So method overloads that involve such comparer interfaces are omitted in this library, or implemented in another form.
- In C#, there are nullable types and default values for a type. For example, `default(int) == 0` and `default(int?) == null`. Some C# methods return such default values if the source sequence is empty, or skip `null`'s if the source sequence contains concrete data too. There are no such notions in Python and the C#-like default semantics are non-existent. So, this usage is not supported by this library (Can `None` be considered a default value for all cases? Hmm..).
- C# has Index syntaxes, and to be Pythonic, these are negative indices. C# has Range, which are slices. This difference can be seen in `Enumerable.element_at()` and `Enumerable.take()`.
- All classes in this library are concrete. There are no interfaces like what are usually done in C#.

Limitations:

- To deal with overloads, some method parameters are positional-only, e.g. those starting with double underscores. Some of them can be improved.

- `OrderedEnumerable` exposing unnecessary type parameter `TKey`.
- `Lookup.__getitem__()`, `Lookup.contains()`, `Lookup.count` are incompatible with the superclass methods they are overriding.

CHAPTER

FOUR

CHANGELOG

GitHub Releases

4.1 v1.2.1

- Fix CachedEnumerable buffering bug when capacity is zero
- Add traverse_topological() to MoreEnumerable class
- Add ranking methods MoreEnumerable.rank() and rank_by() methods
- The documentation page now has hyperlinks for some terminologies in this release

4.2 v1.2.0

- Add pre_scan(), scan(), scan_right() and segment() to MoreEnumerable class
- Fix type annotation mistake in Enumerable.aggregate(__func)
- Fix type annotation mistakes in MoreEnumerable.aggregate_right()

4.3 v1.1.0

- Add consume(), cycle(), and run_length_encode() to MoreEnumerable class
- Fix error in ExtremaEnumerable.take() when it takes a slice

4.4 v1.0.0

- Add enumerate(), rank() and rank_by() to MoreEnumerable class
- Add chunk(), max_by(), min_by(), intersect_by() and union_by() to Enumerable class
- Enumerable.element_at() now supports negative index
- Enumerable.take() now supports taking a slice (which is same as Enumerable.elements_in()) to be consistent with .NET 6
- Enumerable.__getitem__() now supports providing a default value

- **Breaking**: Add Enumerable.distinct_by() that returns an Enumerable instance. MoreEnumerable.distinct_by() that returned a MoreEnumerable instance is removed
- **Breaking**: Add Enumerable.except_by(). The previous MoreEnumerable.except_by() that took homogeneous values as the second iterable is now renamed as MoreEnumerable.except_by2()

4.5 v0.2.1

- Add pipe() to MoreEnumerable class
- Enumerable.distinct(), except1(), .union(), .intersect(), .to_lookup(), .join(), .group_by(), .group_join(), MoreEnumerable.distinct_by(), .except_by() now have preliminary support for unhashable keys

4.6 v0.2.0

- Add a MoreEnumerable class containing the following method names: aggregate_right(), distinct_by(), except_by(), flatten(), for_each(), interleave(), maxima_by(), minima_by(), traverse_breath_first() and traverse_depth_first()
- Add as_more() to Enumerable class

4.7 v0.1.2

- Add to_tuple()
- Add an overload to sequence_equal() that accepts a comparision function
- https://github.com/cleoold/types-linq/commit/f70bd510492a915776f6cac26854111650541b22

4.8 v0.1.1

- Change zip() to support multiple
- Add as_cached() method to memoize results
- Fix OrderedEnumerable bug that once use [] operator on it, returning incorrect result
- Add dunder to some parameter names seen in pyi to prevent them from being passed as named arguments
- https://github.com/cleoold/types-linq/commit/b1b70b9d489cfe06ab1a69c4a0e4a5d195f5f5d7

4.9 v0.1.0

- Initial releases under the BSD-2-Clause License

CHAPTER

FIVE

MODULE TYPES_LINQ.CACHED_ENUMERABLE

5.1 class CachedEnumerable[TSource_co]

```
from types_linq.cached_enumerable import CachedEnumerable
```

Enumerable that stores the enumerated results which can be accessed repeatedly.

Users should not construct instances of this class directly. Use `Enumerable.as_cached()` instead.

Revisions
v0.1.1: New.

5.1.1 Bases

- *Enumerable*[*TSource_co*]

5.1.2 Members

instancemethod as_cached(*, cache_capacity=None)

Parameters
cache_capacity: `Optional[int]`

Returns
CachedEnumerable[*TSource_co*]

Updates settings and returns the original CachedEnumerable reference.

Raises *InvalidOperationError* if cache_capacity is negative.

CHAPTER

SIX

MODULE TYPES_LINQ.ENUMERABLE

6.1 class Enumerable[TSource_co]

```
from types_linq import Enumerable
```

Provides a set of helper methods for querying iterable objects.

6.1.1 Bases

- Sequence[*TSource_co*]
- Generic[*TSource_co*]

6.1.2 Members

instancemethod __init__(__iterable)

Parameters
__iterable: Iterable[*TSource_co*]

Returns
None

Wraps an iterable.

---

instancemethod __init__(__iterable_factory)

Parameters
__iterable_factory: Callable[[], Iterable[*TSource_co*]]

Returns
None

Wraps an iterable returned from the iterable factory. The factory will be called whenever an enumerating operation is performed.

---

instancemethod __contains__(value)

Parameters
value: `object`

Returns
`bool`

Tests whether the sequence contains the specified element. Prefers calling `__contains__()` on the wrapped iterable if available, otherwise, calls `self.contains()`.

Example

```
>>> en = Enumerable([1, 10, 100])
>>> 1000 in en
False
```

---

instancemethod __getitem__(index)

Parameters
index: `int`

Returns
TSource_co

Returns the element at specified index in the sequence. Prefers calling `__getitem__()` on the wrapped iterable if available, otherwise, calls `self.element_at()`.

Example

```
>>> def gen():
...     yield 1; yield 10; yield 100

>>> Enumerable(gen())[1]
10
```

---

instancemethod __getitem__[TDefault](__index_and_default)

Parameters
__index_and_default: `Tuple[int, TDefault]`

Returns
`Union[TSource_co, TDefault]`

Returns the element at specified index in the sequence or returns the default value if it does not exist. Prefers calling `__getitem__()` on the wrapped iterable if available, otherwise, calls `self.element_at()`.

Example

```
>>> def gen():
...     yield 1; yield 10; yield 100

>>> Enumerable(gen())[3, 1000]
1000
```

Revisions
v1.0.0: New.

---

instancemethod `__getitem__(index)`

Parameters
index: `slice`

Returns
Enumerable[*TSource_co*]

Produces a subsequence defined by the given slice notation. Prefers calling `__getitem__()` on the wrapped iterable if available, otherwise, calls `self.elements_in()`.

Example

```
>>> def gen():
...     yield 1; yield 10; yield 100; yield 1000; yield 10000

>>> Enumerable(gen())[1:3].to_list()
[10, 100]
```

---

instancemethod `__iter__()`

Returns
Iterator[*TSource_co*]

Returns an iterator that enumerates the values in the sequence.

Example

```
def gen():
    print('working...')
    yield 1; yield 10; yield 100

query = Enumerable(gen()).select(lambda e: e * 1000)
print('go!')
for e in query:
    print(e)

# output:
# go!
# working...
# 1000
# 10000
# 100000
```

---

instancemethod __len__()

Returns
int

Returns the number of elements in the sequence. Prefers calling __len__() on the wrapped iterable if available, otherwise, calls self.count().

Example

```
>>> en = Enumerable([1, 10, 100])
>>> len(en)
3
```

---

instancemethod __reversed__()

Returns
Iterator[*TSource_co*]

Inverts the order of the elements in the sequence. Prefers calling __reversed__() on the wrapped iterable if available, otherwise, calls self.reverse().

Example

```
>>> ints = [1, 10, 100]
>>> en = Enumerable(ints)
>>> for e in reversed(en):
...     print(e)
100
10
1
```

---

instancemethod aggregate[TAccumulate, TResult](__seed, __func, __result_selector)

Parameters
__seed: *TAccumulate*

__func: Callable[[*TAccumulate*, *TSource_co*], *TAccumulate*]

__result_selector: Callable[[*TAccumulate*], *TResult*]

Returns
TResult

Applies an accumulator function over the sequence. The seed is used as the initial accumulator value, and the result_selector is used to select the result value.

Example

```
>>> fruits = ['apple', 'mango', 'orange', 'passionfruit', 'grape']
>>> Enumerable(fruits).aggregate('banana', lambda acc, e: e if len(e) > len(acc)␣
↪else acc, str.upper)
'PASSIONFRUIT'
```

---

instancemethod aggregate[TAccumulate](__seed, __func)

Parameters
__seed: *TAccumulate*

__func: Callable[[*TAccumulate*, *TSource_co*], *TAccumulate*]

Returns
TAccumulate

Applies an accumulator function over the sequence. The seed is used as the initial accumulator value.

Example

```
>>> words = 'the quick brown fox jumps over the lazy dog'.split(' ')
>>> Enumerable(words).aggregate('end', lambda acc, e: f'{e} {acc}')
'dog lazy the over jumps fox brown quick the end'
```

---

instancemethod aggregate(__func)

Parameters
__func: Callable[[*TSource_co*, *TSource_co*], *TSource_co*]

Returns
TSource_co

Applies an accumulator function over the sequence. Raises *InvalidOperationError* if there is no value in the sequence.

Example

```
>>> words = 'the quick brown fox jumps over the lazy dog'.split(' ')
>>> Enumerable(words).aggregate(lambda acc, e: f'{e} {acc}')
'dog lazy the over jumps fox brown quick the'
```

Example

```
>>> Enumerable.range(1, 10).aggregate(lambda acc, e: acc * e)
3628800
```

Revisions
v1.2.0: Fixed annotation for __func.

---

instancemethod all(predicate)

Parameters
predicate: Callable[[*TSource_co*], bool]

Returns
bool

Tests whether all elements of the sequence satisfy a condition.

Example

```
>>> ints = [1, 3, 5, 7, 9]
>>> Enumerable(ints).all(lambda e: e % 2 == 1)
True
```

---

instancemethod any()

Returns
bool

Tests whether the sequence has any elements.

Example

```
>>> Enumerable([]).any()
False
>>> Enumerable([1]).any()
True
```

---

instancemethod any(__predicate)

Parameters
__predicate: Callable[[*TSource_co*], bool]

Returns
bool

Tests whether any element of the sequence satisfy a condition.

Example

```
>>> ints = [1, 3, 5, 7, 9]
>>> Enumerable(ints).any(lambda e: e % 2 == 0)
False
```

---

instancemethod append(element)

Parameters
element: *TSource_co*

Returns
Enumerable[*TSource_co*]

Appends a value to the end of the sequence. Again, this does not affect the original wrapped object.

Example

```
>>> ints = [1, 3, 5, 7, 9]
>>> Enumerable(ints).append(11).to_list()
[1, 3, 5, 7, 9, 11]
>>> ints
[1, 3, 5, 7, 9]
```

---

instancemethod as_cached(*, cache_capacity=None)

Parameters

cache_capacity: `Optional[int]`

Returns

CachedEnumerable[*TSource_co*]

Returns a CachedEnumerable to cache the enumerated results in this query so that if the wrapped iterable is not repeatable (e.g. generator object), it will be repeatable.

By default, *Enumerable*s constructed from nonrepeatable sources cannot be enumerated multiple times, for example

```
def gen():
    yield 1
    yield 0
    yield 3

query = Enumerable(gen())
print(query.count())
print(query.where(lambda x: x > 0).to_list())
```

prints `3` followed by an empty list `[]`. This is because the `.count()` exhausts the contents in the generator before the second query is run.

To avoid the issue, use this method which saves the results along the way.

```
query = Enumerable(gen()).as_cached()
print(query.count())
print(query.take(2).to_list())
print(query.where(lambda x: x > 0).to_list())
```

printing `3`, `[1, 0]` and `[1, 3]`.

This is an alternative way to deal with non-repeatable sources other than passing function (`query = Enumerable(gen)`) or solidifying the source in advance (`query = Enumerable(list(gen))`). This method is useless if you have constructed an Enumerable from a repeatable source such as a builtin list, an iterable factory mentioned above, or other *Enumerable*'s query methods.

If cache_capacity is None, it is infinite.

Raises *InvalidOperationError* if cache_capacity is negative.

The behavior of this method differs from that of *CachedEnumerable*.

Revisions

v0.1.1: New.

---

instancemethod as_more()

Returns
MoreEnumerable[*TSource_co*]

Returns a MoreEnumerable that has more non-standard query methods available.

Example

```
>>> Enumerable([1, 2, 3]).as_more()
```

Revisions
v0.2.0: New.

---

instancemethod average[TResult]()

Constraint
self: *Enumerable*[*SupportsAverage*[*TResult*]]

Returns
TResult

Computes the average value of the sequence. Raises *InvalidOperationError* if there is no value.

Example

```
>>> ints = [1, 3, 5, 9, 11]
>>> Enumerable(ints).average()
5.8
```

---

instancemethod average[TResult](__selector)

Parameters
__selector: Callable[[*TSource_co*], *SupportsAverage*[*TResult*]]

Returns
TResult

Computes the average value of the sequence using the selector. Raises *InvalidOperationError* if there is no value.

Example

```
>>> strs = ['1', '3', '5', '9', '11']
>>> Enumerable(strs).average(lambda e: int(e) * 1000)
5800.0
```

---

instancemethod average2[TResult, TDefault](__default)

Constraint
self: *Enumerable*[*SupportsAverage*[*TResult*]]

Parameters
__default: *TDefault*

Returns
Union[*TResult*, *TDefault*]

Computes the average value of the sequence. Returns `default` if there is no value.

Example

```
>>> Enumerable([1, 2]).average2(0)
1.5
>>> Enumerable([]).average2(0)
0
```

---

instancemethod average2[TResult, TDefault](__selector, __default)

Parameters
__selector: Callable[[*TSource_co*], *SupportsAverage*[*TResult*]]

__default: *TDefault*

Returns
Union[*TResult*, *TDefault*]

Computes the average value of the sequence using the selector. Returns `default` if there is no value.

Example

```
>>> Enumerable([]).average2(lambda e: int(e) * 1000, 0)
0
```

---

instancemethod cast[TResult](__t_result)

Parameters
__t_result: Type[*TResult*]

Returns
Enumerable[*TResult*]

Casts the elements to the specified type.

This method does not change anything. It returns the original Enumerable reference unchanged.

Example

```
query: Enumerable[object] = ...
same_query: Enumerable[int] = query.cast(int)
```

---

instancemethod chunk(size)

Parameters
size: int

Returns
Enumerable[MutableSequence[*TSource_co*]]

Splits the elements of a sequence into chunks of size at most the provided size. Raises *InvalidOperationError* if size is less than 1.

Example

```
>>> def source(i):
...     while True:
...         yield i
...         i *= 3

>>> en = Enumerable(source(1)).chunk(4).take(3)
>>> for chunk in en:
...     print(chunk)
[1, 3, 9, 27]
[81, 243, 729, 2187]
[6561, 19683, 59049, 177147]
```

Revisions
v1.0.0: New.

---

instancemethod concat(second)

Parameters
second: Iterable[*TSource_co*]

Returns
Enumerable[*TSource_co*]

Concatenates two sequences.

Example

```
>>> en1 = Enumerable([1, 2, 3])
>>> en2 = Enumerable([1, 2, 4])
>>> en1.concat(en2).to_list()
[1, 2, 3, 1, 2, 4]
```

---

instancemethod contains(value)

Parameters
value: object

Returns
bool

Tests whether the sequence contains the specified element using ==.

This method always uses a generic element-finding method (O(n)) regardless the implementation of the wrapped iterable.

Example

```
>>> def gen():
...     yield 1; yield 10; yield 100

>>> Enumerable(gen()).contains(11)
False
```

---

instancemethod contains[TOther](value, __comparer)

Parameters
value: *TOther*

__comparer: Callable[[*TSource_co*, *TOther*], bool]

Returns
bool

Tests whether the sequence contains the specified element using the provided comparer that returns True if two values are equal.

Example

```
>>> ints = [1, 3, 5, 7, 9]
>>> Enumerable(ints).contains('9', lambda x, y: str(x) == y)
True
```

---

instancemethod count()

Returns
int

Returns the number of elements in the sequence.

This method always uses a generic length-finding method (O(n)) regardless the implementation of the wrapped iterable.

Example

```
>>> def gen():
...     yield 1; yield 10; yield 100

>>> Enumerable(gen()).count()
3
```

---

instancemethod count(__predicate)

Parameters
__predicate: Callable[[*TSource_co*], bool]

Returns
int

Returns the number of elements that satisfy the condition.

Example

```
>>> def gen():
...     yield 1; yield 10; yield 100

>>> Enumerable(gen()).count(lambda e: e % 10 == 0)
2
```

---

instancemethod default_if_empty[TDefault](default)

Parameters
default: *TDefault*

Returns
Union[*Enumerable*[*TSource_co*], *Enumerable*[*TDefault*]]

Returns the elements of the sequence or the provided value in a singleton collection if the sequence is empty.

Example

```
>>> Enumerable([]).default_if_empty(0).to_list()
[0]
>>> Enumerable([44, 45, 56]).default_if_empty(0).to_list()
[44, 45, 56]
```

---

instancemethod `distinct()`

Returns
Enumerable[*TSource_co*]

Returns distinct elements from the sequence.

Example

```
>>> ints = [1, 4, 5, 6, 4, 3, 1, 99]
>>> Enumerable(ints).distinct().to_list()
[1, 4, 5, 6, 3, 99]
```

Revisions
v0.2.1: Added preliminary support for unhashable values.

---

instancemethod `distinct_by(key_selector)`

Parameters
key_selector: `Callable[[TSource_co], object]`

Returns
Enumerable[*TSource_co*]

Returns distinct elements from the sequence where "distinctness" is determined by the value returned by the selector.

Example

```
>>> ints = [1, 4, 5, 6, 4, 3, 1, 99]
>>> Enumerable(ints).distinct_by(lambda x: x // 2).to_list()
[1, 4, 6, 3, 99]
```

Revisions
v1.0.0: New. The method with same name (but different return type) in MoreEnumerable class was removed.

---

instancemethod `element_at(index)`

Parameters
index: `int`

Returns
TSource_co

Returns the element at specified index in the sequence. *IndexOutOfRangeError* is raised if no such element exists.

If the index is negative, it means counting from the end.

This method always uses a generic list element-finding method (O(n)) regardless the implementation of the wrapped iterable.

Example

```
>>> def gen():
...     yield 1; yield 10; yield 100

>>> Enumerable(gen()).element_at(1)
10

>>> Enumerable(gen()).element_at(-1)
100
```

Revisions

v1.0.0: Added support for negative index.

---

instancemethod element_at[TDefault](index, __default)

Parameters

index: int

__default: *TDefault*

Returns

Union[*TSource_co*, *TDefault*]

Returns the element at specified index in the sequence. Default value is returned if no such element exists.

If the index is negative, it means counting from the end.

This method always uses a generic list element-finding method (O(n)) regardless the implementation of the wrapped iterable.

Example

```
>>> def gen():
...     yield 1; yield 10; yield 100

>>> Enumerable(gen()).element_at(3, 0)
0
```

Revisions

v1.0.0: Added support for negative index.

---

staticmethod empty()

Returns

Enumerable[*TSource_co*]

Returns an empty enumerable.

Example

```
>>> en := Enumerable.empty()
<types_linq.enumerable.Enumerable at 0x00000000000>
>>> en.to_list()
[]
```

instancemethod except1(second)

Parameters

second: Iterable[*TSource_co*]

Returns

Enumerable[*TSource_co*]

Produces the set difference of two sequences: self - second.

Note `except` is a keyword in Python.

Example

```
>>> ints = [1, 2, 3, 4, 5]
>>> Enumerable(ints).except1([1, 3, 5, 7, 9]).to_list()
[2, 4]
```

Revisions

v0.2.1: Added preliminary support for unhashable values.

---

instancemethod except_by[TKey](second, key_selector)

Parameters

second: Iterable[*TKey*]

key_selector: Callable[[*TSource_co*], *TKey*]

Returns

Enumerable[*TSource_co*]

Produces the set difference of two sequences: self - second, according to a key selector that determines "distinctness".

Example

```
>>> first = [(16, 'x'), (9, 'y'), (12, 'd'), (16, 't')]
>>> second = ['y', 'd']
>>> Enumerable(first).except_by(second, lambda x: x[1]).to_list()
[(16, 'x'), (16, 't')]
```

Revisions

v1.0.0: New. The method with same name (but different usage) in MoreEnumerable class was renamed as `except_by2()` to accommodate this.

---

instancemethod first()

Returns
TSource_co

Returns the first element of the sequence. Raises *InvalidOperationError* if there is no first element.

This method always uses a generic method to enumerate the first element regardless the implementation of the wrapped iterable.

Example

```
>>> def gen():
...     yield 1; yield 10; yield 100

>>> Enumerable(gen()).first()
1
```

---

instancemethod first(__predicate)

Parameters
__predicate: Callable[[*TSource_co*], bool]

Returns
TSource_co

Returns the first element of the sequence that satisfies the condition. Raises *InvalidOperationError* if no such element exists.

Example

```
>>> ints = [1, 3, 5, 7, 9, 11, 13]
>>> Enumerable(ints).first(lambda e: e > 10)
11
```

---

instancemethod first2[TDefault](__default)

Parameters
__default: *TDefault*

Returns
Union[*TSource_co*, *TDefault*]

Returns the first element of the sequence or a default value if there is no such element.

This method always uses a generic method to enumerate the first element regardless the implementation of the wrapped iterable.

Example

```
>>> def gen(ok: bool):
...     if ok:
...         yield 1; yield 10; yield 100

>>> Enumerable(gen(True)).first2(0)
1
>>> Enumerable(gen(False)).first2(0)
0
```

---

instancemethod first2[TDefault](__predicate, __default)

Parameters

__predicate: Callable[[*TSource_co*], bool]

__default: *TDefault*

Returns

Union[*TSource_co*, *TDefault*]

Returns the first element of the sequence that satisfies the condition or a default value if no such element exists.

Example

```
>>> ints = [1, 3, 5, 7, 9, 11, 13]
>>> Enumerable(ints).first2(lambda e: e > 100, 100)
100
```

---

instancemethod group_by[TKey, TValue, TResult](key_selector, value_selector, __result_selector)

Parameters

key_selector: Callable[[*TSource_co*], *TKey*]

value_selector: Callable[[*TSource_co*], *TValue*]

__result_selector: Callable[[*TKey*, *Enumerable*[*TValue*]], *TResult*]

Returns

Enumerable[*TResult*]

Groups the elements of the sequence according to specified key selector and value selector. Then it returns the result value using each grouping and its key.

Example

```
>>> pets_list = [
...     ('Barley', 8.3), ('Boots', 4.9), ('Whiskers', 1.5), ('Daisy', 4.3),
...     ('Roman', 8.6), ('Fangus', 8.6), ('Roam', 2.2), ('Roll', 1.4),
... ]

>>> en = Enumerable(pets_list).group_by(
...     lambda pet: math.floor(pet[1]),
```

(continues on next page)

(continued from previous page)

```
...     lambda pet: pet[0],
...     lambda age_floored, names: (age_floored, names.to_set()),
... )

>>> for obj in en:
...     print(obj)
(8, {'Fangus', 'Roman', 'Barley'})
(4, {'Boots', 'Daisy'})
(1, {'Roll', 'Whiskers'})
(2, {'Roam'})
```

Revisions

v0.2.1: Added preliminary support for unhashable keys.

---

instancemethod group_by[TKey, TValue](key_selector, value_selector)

Parameters

key_selector: `Callable[[TSource_co], TKey]`

value_selector: `Callable[[TSource_co], TValue]`

Returns

Enumerable[*Grouping*[*TKey*, *TValue*]]

Groups the elements of the sequence according to specified key selector and value selector.

Example

```
>>> en = Enumerable(pets_list).group_by(
...     lambda pet: math.floor(pet[1]),
...     lambda pet: pet[0],
... )

>>> for grouping in en:
...     print(grouping.key, grouping.to_set())
8 {'Fangus', 'Roman', 'Barley'}
4 {'Boots', 'Daisy'}
1 {'Roll', 'Whiskers'}
2 {'Roam'}
```

Revisions

v0.2.1: Added preliminary support for unhashable keys.

---

instancemethod group_by2[TKey, TResult](key_selector, __result_selector)

Parameters

key_selector: `Callable[[TSource_co], TKey]`

__result_selector: `Callable[[TKey, Enumerable[TSource_co]], TResult]`

Returns

Enumerable[*TResult*]

Groups the elements of the sequence according to a specified key selector function and creates a result value using each grouping and its key.

Example

```
>>> en = Enumerable(pets_list).group_by2(
...     lambda pet: math.floor(pet[1]),
...     lambda age_floored, pets: (age_floored, pets.to_list()),
... )

>>> for obj in en:
...     print(obj)
(8, [('Barley', 8.3), ('Roman', 8.6), ('Fangus', 8.6)])
(4, [('Boots', 4.9), ('Daisy', 4.3)])
(1, [('Whiskers', 1.5), ('Roll', 1.4)])
(2, [('Roam', 2.2)])
```

Revisions

v0.2.1: Added preliminary support for unhashable keys.

---

instancemethod group_by2[TKey](key_selector)

Parameters

key_selector: `Callable[[TSource_co], TKey]`

Returns

Enumerable[*Grouping*[*TKey*, *TSource_co*]]

Groups the elements of the sequence according to a specified key selector function.

Example

```
>>> en = Enumerable(pets_list).group_by2(
...     lambda pet: math.floor(pet[1]),
... )

>>> for grouping in en:
...     print(grouping.key, grouping.to_list())
8 [('Barley', 8.3), ('Roman', 8.6), ('Fangus', 8.6)]
4 [('Boots', 4.9), ('Daisy', 4.3)]
1 [('Whiskers', 1.5), ('Roll', 1.4)]
2 [('Roam', 2.2)]
```

Revisions

v0.2.1: Added preliminary support for unhashable keys.

instancemethod group_join[TInner, TKey, TResult](inner, outer_key_selector, inner_key_selector, result_selector)

Parameters

inner: Iterable[*TInner*]

outer_key_selector: Callable[[*TSource_co*], *TKey*]

inner_key_selector: Callable[[*TInner*], *TKey*]

result_selector: Callable[[*TSource_co*, *Enumerable*[*TInner*]], *TResult*]

Returns

Enumerable[*TResult*]

Correlates the elements of two sequences based on equality of keys and groups the results using the selector.

In normal cases, the iteration preserves order of elements in self (outer), and for each element in self, the order of matching elements from inner.

Unhashable keys are supported (where hashibility is determined by checking `typing.Hashable`). If any keys formed by key selectors involve such types, the order is unspecified.

Example

```
>>> class Person(NamedTuple):
...     name: str
>>> class Pet(NamedTuple):
...     name: str
...     owner: Person

>>> magnus = Person('Hedlund, Magnus')
>>> terry = Person('Adams, Terry')
>>> charlotte = Person('Weiss, Charlotte')
>>> poor = Person('Animal, No')
>>> barley = Pet('Barley', owner=terry)
>>> boots = Pet('Boots', owner=terry)
>>> whiskers = Pet('Whiskers', owner=charlotte)
>>> daisy = Pet('Daisy', owner=magnus)
>>> roman = Pet('Roman', owner=terry)

>>> people = [magnus, terry, charlotte, poor]
>>> pets = [barley, boots, whiskers, daisy, roman]

>>> en = Enumerable(people).group_join(
...     pets,
...     lambda person: person,
...     lambda pet: pet.owner,
...     lambda person, pet_collection: (
...         person.name,
...         pet_collection.select(lambda pet: pet.name).to_set(),
...     ),
... )
```

(continues on next page)

(continued from previous page)

```
>>> for obj in en:
...     print(obj)
('Hedlund, Magnus', {'Daisy'})
('Adams, Terry', {'Boots', 'Roman', 'Barley'})
('Weiss, Charlotte', {'Whiskers'})
('Animal, No', set())
```

Revisions

v0.2.1: Added preliminary support for unhashable keys.

---

instancemethod `intersect(second)`

Parameters

second: `Iterable[`*TSource_co*`]`

Returns

Enumerable`[`*TSource_co*`]`

Produces the set intersection of two sequences: self & second.

Example

```
>>> ints = [1, 3, 5, 7, 9, 11]
>>> Enumerable(ints).intersect([1, 2, 3, 4, 5]).to_list()
[1, 3, 5]
```

Revisions

v0.2.1: Added preliminary support for unhashable values.

---

instancemethod `intersect_by[TKey](second, key_selector)`

Parameters

second: `Iterable[`*TKey*`]`

key_selector: `Callable[[`*TSource_co*`], `*TKey*`]`

Returns

Enumerable`[`*TSource_co*`]`

Produces the set intersection of two sequences: self & second according to a specified key selector.

Example

```
>>> strs = ['+1', '-3', '+5', '-7', '+9', '-11']
>>> Enumerable(strs).intersect_by([1, 2, 3, 5, 9], lambda x: abs(int(x))).to_list()
['+1', '-3', '+5', '+9']
```

Revisions

v1.0.0: New.

---

instancemethod join[TInner, TKey, TResult](inner, outer_key_selector, inner_key_selector, result_selector)

Parameters

inner: Iterable[*TInner*]

outer_key_selector: Callable[[*TSource_co*], *TKey*]

inner_key_selector: Callable[[*TInner*], *TKey*]

result_selector: Callable[[*TSource_co*, *TInner*], *TResult*]

Returns

Enumerable[*TResult*]

Correlates the elements of two sequences based on matching keys.

In normal cases, the iteration preserves order of elements in self (outer), and for each element in self, the order of matching elements from inner.

Unhashable keys are supported (where hashibility is determined by checking `typing.Hashable`). If any keys formed by key selectors involve such types, the order is unspecified.

Example

```
# Please refer to group_join() for definition of people and pets

>>> en = Enumerable(people).join(
...     pets,
...     lambda person: person,
...     lambda pet: pet.owner,
...     lambda person, pet: (person.name, pet.name),
... )

>>> for obj in en:
...     print(obj)
('Hedlund, Magnus', 'Daisy')
('Adams, Terry', 'Barley')
('Adams, Terry', 'Boots')
('Adams, Terry', 'Roman')
('Weiss, Charlotte', 'Whiskers')
```

Revisions

v0.2.1: Added preliminary support for unhashable keys.

---

instancemethod last()

Returns

TSource_co

Returns the last element of the sequence. Raises *InvalidOperationError* if there is no first element.

This method always uses a generic method to enumerate the last element (O(n)) regardless the implementation of the wrapped iterable.

Example

```
>>> def gen():
...     yield 1; yield 10; yield 100

>>> Enumerable(gen()).last()
100
```

---

instancemethod `last(__predicate)`

Parameters
__predicate: `Callable[[TSource_co], bool]`

Returns
TSource_co

Returns the last element of the sequence that satisfies the condition. Raises *InvalidOperationError* if no such element exists.

Example

```
>>> ints = [1, 3, 5, 7, 9, 11, 13]
>>> Enumerable(ints).last(lambda e: e < 10)
9
```

---

instancemethod `last2[TDefault](__default)`

Parameters
__default: *TDefault*

Returns
`Union[TSource_co, TDefault]`

Returns the last element of the sequence or a default value if there is no such element.

This method always uses a generic method to enumerate the last element (O(n)) regardless the implementation of the wrapped iterable.

Example

```
>>> def gen(ok: bool):
...     if ok:
...         yield 1; yield 10; yield 100

>>> Enumerable(gen(True)).last2(9999)
100
>>> Enumerable(gen(False)).last2(9999)
9999
```

---

instancemethod last2[TDefault](__predicate, __default)

Parameters

__predicate: Callable[[*TSource_co*], bool]

__default: *TDefault*

Returns

Union[*TSource_co*, *TDefault*]

Returns the last element of the sequence that satisfies the condition or a default value if no such element exists.

Example

```
>>> ints = [13, 11, 9, 7, 5, 3, 1]
>>> Enumerable(ints).last2(lambda e: e < 0, 9999)
9999
```

---

instancemethod max[TSupportsLessThan]()

Constraint

self: *Enumerable*[*TSupportsLessThan*]

Returns

TSupportsLessThan

Returns the maximum value in the sequence. Raises *InvalidOperationError* if there is no value.

Example

```
>>> nums = [1, 5, 2.2, 5, 1, 2]
>>> Enumerable(nums).max()
5
```

---

instancemethod max[TSupportsLessThan](__result_selector)

Parameters

__result_selector: Callable[[*TSource_co*], *TSupportsLessThan*]

Returns

TSupportsLessThan

Invokes a transform function on each element of the sequence and returns the maximum of the resulting values. Raises *InvalidOperationError* if there is no value.

Example

```
>>> strs = ['aaa', 'bb', 'c', 'dddd']
>>> Enumerable(strs).max(len)
4
```

---

instancemethod max2[TSupportsLessThan, TDefault](__default)

Constraint
self: *Enumerable*[*TSupportsLessThan*]

Parameters
__default: *TDefault*

Returns
Union[*TSupportsLessThan*, *TDefault*]

Returns the maximum value in the sequence, or the default one if there is no value.

Example

```
>>> Enumerable([]).max2(0)
0
```

---

instancemethod max2[TSupportsLessThan, TDefault](__result_selector, __default)

Parameters
__result_selector: Callable[[*TSource_co*], *TSupportsLessThan*]

__default: *TDefault*

Returns
Union[*TSupportsLessThan*, *TDefault*]

Invokes a transform function on each element of the sequence and returns the maximum of the resulting values. Returns the default one if there is no value.

Example

```
>>> Enumerable([]).max2(len, 0)
0
>>> Enumerable(['a']).max2(len, 0)
1
```

---

instancemethod max_by[TSupportsLessThan](key_selector)

Parameters
key_selector: Callable[[*TSource_co*], *TSupportsLessThan*]

Returns
TSource_co

Returns the maximal element of the sequence based on the given key selector. Raises *InvalidOperationError* if there is no value.

Example

```
>>> strs = ['aaa', 'bb', 'c', 'dddd']
>>> Enumerable(strs).max_by(len)
'dddd'
```

Revisions
v1.0.0: New.

---

instancemethod `max_by[TKey](key_selector, __comparer)`

Parameters
key_selector: `Callable[[TSource_co], TKey]`

__comparer: `Callable[[TKey, TKey], int]`

Returns
TSource_co

Returns the maximal element of the sequence based on the given key selector and the comparer. Raises *InvalidOperationError* if there is no value.

Such comparer takes two values and return positive ints when lhs > rhs, negative ints if lhs < rhs, and 0 if they are equal.

Revisions
v1.0.0: New.

---

instancemethod `min[TSupportsLessThan]()`

Constraint
self: *Enumerable*[*TSupportsLessThan*]

Returns
TSupportsLessThan

Returns the minimum value in the sequence. Raises *InvalidOperationError* if there is no value.

---

instancemethod `min[TSupportsLessThan](__result_selector)`

Parameters
__result_selector: `Callable[[TSource_co], TSupportsLessThan]`

Returns
TSupportsLessThan

Invokes a transform function on each element of the sequence and returns the minimum of the resulting values. Raises *InvalidOperationError* if there is no value.

---

instancemethod min2[TSupportsLessThan, TDefault](__default)

Constraint
self: *Enumerable*[*TSupportsLessThan*]

Parameters
__default: *TDefault*

Returns
Union[*TSupportsLessThan*, *TDefault*]

Returns the minimum value in the sequence, or the default one if there is no value.

---

instancemethod min2[TSupportsLessThan, TDefault](__result_selector, __default)

Parameters
__result_selector: Callable[[*TSource_co*], *TSupportsLessThan*]

__default: *TDefault*

Returns
Union[*TSupportsLessThan*, *TDefault*]

Invokes a transform function on each element of the sequence and returns the minimum of the resulting values. Returns the default one if there is no value.

---

instancemethod min_by[TSupportsLessThan](key_selector)

Parameters
key_selector: Callable[[*TSource_co*], *TSupportsLessThan*]

Returns
TSource_co

Returns the minimal element of the sequence based on the given key selector. Raises *InvalidOperationError* if there is no value.

Revisions
v1.0.0: New.

---

instancemethod min_by[TKey](key_selector, __comparer)

Parameters
key_selector: Callable[[*TSource_co*], *TKey*]

__comparer: Callable[[*TKey*, *TKey*], int]

Returns
TSource_co

Returns the minimal element of the sequence based on the given key selector and the comparer. Raises *InvalidOperationError* if there is no value.

Such comparer takes two values and return positive ints when lhs > rhs, negative ints if lhs < rhs, and 0 if they are equal.

Revisions
v1.0.0: New.

---

instancemethod `of_type[TResult](t_result)`

Parameters
t_result: `Type[TResult]`

Returns
Enumerable`[TResult]`

Filters elements based on the specified type.

Builtin `isinstance()` is used.

Example

```
>>> lst = [1, 14, object(), True, []]
>>> Enumerable(lst).of_type(int).to_list()
[1, 14, True]
```

---

instancemethod `order_by[TSupportsLessThan](key_selector)`

Parameters
key_selector: `Callable[[TSource_co], TSupportsLessThan]`

Returns
OrderedEnumerable`[TSource_co, TSupportsLessThan]`

Sorts the elements of the sequence in ascending order according to a key.

Example

```
>>> ints = [8, 4, 5, 2]
>>> Enumerable(ints).order_by(lambda e: e).to_list()
[2, 4, 5, 8]
```

Example

```
>>> class Pet(NamedTuple):
...     name: str
...     age: int

>>> pets = [Pet('Barley', 8), Pet('Boots', 4), Pet('Roman', 5)]
>>> Enumerable(pets).order_by(lambda p: p.age) \
...     .select(lambda p: p.name)              \
...     .to_list()
['Boots', 'Roman', 'Barley']
```

Subsequent ordering is supported. See *OrderedEnumerable*.

---

instancemethod order_by[TKey](key_selector, __comparer)

Parameters

key_selector: Callable[[*TSource_co*], *TKey*]

__comparer: Callable[[*TKey*, *TKey*], int]

Returns

OrderedEnumerable[*TSource_co*, *TKey*]

Sorts the elements of the sequence in ascending order by using a specified comparer.

Such comparer takes two values and return positive ints when lhs > rhs, negative ints if lhs < rhs, and 0 if they are equal. In fact, this overload should not be used (see Sorting HOW TO).

Example

```
>>> Enumerable(pets).order_by(lambda p: p, lambda pl, pr: pl.age - pr.age) \
...     .select(lambda p: p.name)                                          \
...     .to_list()
['Boots', 'Roman', 'Barley']
```

---

instancemethod order_by_descending[TSupportsLessThan](key_selector)

Parameters

key_selector: Callable[[*TSource_co*], *TSupportsLessThan*]

Returns

OrderedEnumerable[*TSource_co*, *TSupportsLessThan*]

Sorts the elements of the sequence in descending order according to a key.

Example

```
>>> ints = [8, 4, 5, 2]
>>> Enumerable(ints).order_by_descending(lambda e: e).to_list()
[8, 5, 4, 2]
```

---

instancemethod order_by_descending[TKey](key_selector, __comparer)

Parameters

key_selector: Callable[[*TSource_co*], *TKey*]

__comparer: Callable[[*TKey*, *TKey*], int]

Returns

OrderedEnumerable[*TSource_co*, *TKey*]

Sorts the elements of the sequence in descending order by using a specified comparer.

Such comparer takes two values and return positive ints when lhs > rhs, negative ints if lhs < rhs, and 0 if they are equal.

---

instancemethod prepend(element)

Parameters
element: *TSource_co*

Returns
Enumerable[*TSource_co*]

Adds a value to the beginning of the sequence. Again, this does not affect the original wrapped object.

Example

```
>>> ints = [1, 3, 5, 7, 9]
>>> Enumerable(ints).prepend(-1).to_list()
[-1, 1, 3, 5, 7, 9]
```

---

staticmethod range(start, count)

Parameters
start: int

count: Optional[int]

Returns
Enumerable[int]

Generates a sequence of count integral numbers from start, incrementing each by one.

If count is None, the sequence is infinite. Raises *InvalidOperationError* if count is negative.

Example

```
>>> Enumerable.range(-5, 6).to_list()
[-5, -4, -3, -2, -1, 0]
```

---

staticmethod repeat[TResult](value, count=None)

Parameters
value: *TResult*

count: Optional[int]

Returns
Enumerable[*TResult*]

Generates a sequence that contains one repeated value.

If count is None, the sequence is infinite. Raises *InvalidOperationError* if count is negative.

Example

```
>>> Enumerable.repeat(0, 6).to_list()
[0, 0, 0, 0, 0, 0]
```

---

instancemethod reverse()

Returns
Enumerable[*TSource_co*]

Inverts the order of the elements in the sequence.

This method always uses a generic reverse traversal method regardless the implementation of the wrapped iterable.

Example

```
>>> def gen():
...     yield 1; yield 10; yield 100

>>> Enumerable(gen()).reverse().to_list()
[100, 10, 1]
```

---

instancemethod select[TResult](selector)

Parameters
selector: Callable[[*TSource_co*], *TResult*]

Returns
Enumerable[*TResult*]

Projects each element of the sequence into a new form.

Example

```
>>> ints = [1, 3, 5, 7, 9]
>>> Enumerable(ints).select(lambda e: '*' * e).to_list()
['*', '***', '*****', '*******', '*********']
```

---

instancemethod select2[TResult](selector)

Parameters
selector: Callable[[*TSource_co*, int], *TResult*]

Returns
Enumerable[*TResult*]

Projects each element of the sequence into a new form by incorporating the indices.

Example

```
>>> ints = [1, 3, 5, 7, 9]
>>> Enumerable(ints).select2(lambda e, i: e * (i + 1)).to_list()
[1, 6, 15, 28, 45]
```

---

instancemethod select_many[TCollection, TResult](collection_selector, __result_selector)

Parameters

collection_selector: Callable[[*TSource_co*], Iterable[*TCollection*]]

__result_selector: Callable[[*TSource_co*, *TCollection*], *TResult*]

Returns

Enumerable[*TResult*]

Projects each element of the sequence into an iterable, flattens the resulting sequence into one sequence, then calls result_selector on each element therein.

Example

```
>>> pet_owners = [
...     {'name': 'Higa', 'pets': ['Scruffy', 'Sam']},
...     {'name': 'Ashkenazi', 'pets': ['Walker', 'Sugar']},
...     {'name': 'Hines',  'pets': ['Dusty']},
... ]

>>> en = Enumerable(pet_owners).select_many(
...     lambda owner: owner['pets'],
...     lambda owner, name: (name, owner['name']),
... )

>>> for tup in en:
...     print(tup)
('Scruffy', 'Higa')
('Sam', 'Higa')
('Walker', 'Ashkenazi')
('Sugar', 'Ashkenazi')
('Dusty', 'Hines')
```

---

instancemethod select_many[TResult](__selector)

Parameters

__selector: Callable[[*TSource_co*], Iterable[*TResult*]]

Returns

Enumerable[*TResult*]

Projects each element of the sequence to an iterable and flattens the resultant sequences.

Example

```
>>> sentences = ['i select things', 'i do many times']
>>> Enumerable(sentences).select_many(str.split).to_list()
['i', 'select', 'things', 'i', 'do', 'many', 'times']
```

---

instancemethod select_many2[TCollection, TResult](collection_selector, __result_selector)

Parameters

collection_selector: Callable[[*TSource_co*, int], Iterable[*TCollection*]]

__result_selector: Callable[[*TSource_co*, *TCollection*], *TResult*]

Returns

Enumerable[*TResult*]

Projects each element of the sequence into an iterable, flattens the resulting sequence into one sequence, then calls result_selector on each element therein. The indices of source elements are used.

---

instancemethod select_many2[TResult](__selector)

Parameters

__selector: Callable[[*TSource_co*, int], Iterable[*TResult*]]

Returns

Enumerable[*TResult*]

Projects each element of the sequence to an iterable and flattens the resultant sequences. The indices of source elements are used.

Example

```
>>> dinner = ['Ramen with Egg and Beef', 'Gyoza', 'Fried Chicken']
>>> en = Enumerable(dinner).select_many2(
...     lambda e, i: Enumerable(e.split(' '))
...         .where(lambda w: w[0].isupper())
...         .select(lambda w: f'Table {i}: {w}'),
... )
>>> for s in en:
...     print(s)
Table 0: Ramen
Table 0: Egg
Table 0: Beef
Table 1: Gyoza
Table 2: Fried
Table 2: Chicken
```

---

instancemethod sequence_equal(second)

Parameters
second: Iterable[*TSource_co*]

Returns
bool

Determines whether two sequences are equal using == on each element.

Example

```
>>> def gen():
...     yield 1; yield 10; yield 100
>>> lst = [1, 10, 100]

>>> Enumerable(gen()).sequence_equal(lst)
True
```

---

instancemethod sequence_equal[TOther](second, __comparer)

Parameters
second: Iterable[*TOther*]

__comparer: Callable[[*TSource_co*, *TOther*], bool]

Returns
bool

Determines whether two sequences are equal using a comparer that returns True if two values are equal, on each element.

Example

```
>>> ints = [1, 3, 5, 7, 9]
>>> strs = ['1', '3', '5', '7', '9']
>>> Enumerable(ints).sequence_equal(strs, lambda x, y: str(x) == y)
True
```

Revisions
v0.1.2: New.

---

instancemethod single()

Returns
TSource_co

Returns the only element in the sequence. Raises *InvalidOperationError* if the sequence does not contain exactly one element.

Example

```
>>> Enumerable([5]).single()
5
```

Example

```
>>> lst = [5, 6]
>>> try:
...     print(Enumerable(lst).single())
... except InvalidOperationError:
...     print('Collection does not contain exactly one element. Sorry.')
Collection does not contain exactly one element. Sorry.
```

---

instancemethod single(__predicate)

Parameters

__predicate: Callable[[*TSource_co*], bool]

Returns

TSource_co

Returns the only element in the sequence that satisfies the condition. Raises *InvalidOperationError* if no element satisfies the condition, or more than one do.

Example

```
>>> ints = [1, 3, 5, 7, 9, 11, 9]
>>> Enumerable(ints).single(lambda e: e > 10)
11
>>> try:
...     Enumerable(ints).single(lambda e: e == 9)
... except InvalidOperationError:
...     print('Too many nines!')
Too many nines!
```

---

instancemethod single2[TDefault](__default)

Parameters

__default: *TDefault*

Returns

Union[*TSource_co*, *TDefault*]

Returns the only element in the sequence or the default value if the sequence is empty. Raises *InvalidOperationError* if there are more than one elements in the sequence.

Example

```
>>> Enumerable([]).single2(0)
0
```

---

instancemethod single2[TDefault](__predicate, __default)

Parameters

__predicate: Callable[[*TSource_co*], bool]

__default: *TDefault*

Returns

Union[*TSource_co*, *TDefault*]

Returns the only element in the sequence that satisfies the condition, or the default value if there is no such element. Raises *InvalidOperationError* if there are more than one elements satisfying the condition.

Example

```
>>> fruits = ['apple', 'banana', 'mango']
>>> Enumerable(fruits).single2(lambda e: len(e) > 10, 'sorry')
'sorry'
```

---

instancemethod skip(count)

Parameters

count: int

Returns

Enumerable[*TSource_co*]

Bypasses a specified number of elements in the sequence and then returns the remaining.

Example

```
>>> grades = [59, 82, 70, 56, 92, 98, 85]
>>> Enumerable(grades).order_by_descending(lambda g: g).skip(3).to_list()
[82, 70, 59, 56]
```

---

instancemethod skip_last(count)

Parameters

count: int

Returns

Enumerable[*TSource_co*]

Returns a new sequence that contains the elements of the current sequence with last `count` elements omitted.

Example

```
>>> grades = [59, 82, 70, 56, 92, 98, 85]
>>> Enumerable(grades).order_by_descending(lambda g: g).skip_last(3).to_list()
[98, 92, 85, 82]
```

---

instancemethod skip_while(predicate)

Parameters
predicate: Callable[[*TSource_co*], bool]

Returns
Enumerable[*TSource_co*]

Bypasses elements in the sequence as long as the condition is true and then returns the remaining elements.

Example

```
>>> grades = [59, 82, 70, 56, 92, 98, 85]
>>> Enumerable(grades).order_by_descending(lambda g: g) \
...     .skip_while(lambda g: g >= 80)                  \
...     .to_list()
[70, 59, 56]
```

---

instancemethod skip_while2(predicate)

Parameters
predicate: Callable[[*TSource_co*, int], bool]

Returns
Enumerable[*TSource_co*]

Bypasses elements in the sequence as long as the condition is true and then returns the remaining elements. The element's index is used in the predicate function.

Example

```
>>> amounts = [500, 250, 900, 800, 650, 400, 150, 550]
>>> Enumerable(amounts).skip_while2(lambda a, i: a > i * 100).to_list()
[400, 150, 550]
```

---

instancemethod sum[TSupportsAdd]()

Constraint
self: *Enumerable*[*TSupportsAdd*]

Returns
Union[*TSupportsAdd*, int]

Computes the sum of the sequence, or `0` if the sequence is empty.

Example

```
>>> floats = [.1, .3, .5, .9, 1.1]
>>> Enumerable(floats).sum()
2.9000000000000004
```

---

instancemethod sum[TSupportsAdd](__selector)

Parameters
__selector: Callable[[*TSource_co*], *TSupportsAdd*]

Returns
Union[*TSupportsAdd*, int]

Computes the sum of the sequence using the selector. Returns `0` if the sequence is empty.

Example

```
>>> floats = [.1, .3, .5, .9, 1.1]
>>> Enumerable(floats).sum(lambda e: int(e * 1000))
2900
```

---

instancemethod sum2[TSupportsAdd, TDefault](__default)

Constraint
self: *Enumerable*[*TSupportsAdd*]

Parameters
__default: *TDefault*

Returns
Union[*TSupportsAdd*, *TDefault*]

Computes the sum of the sequence. Returns the default value if it is empty.

Example

```
>>> Enumerable([]).sum2(880)
880
```

---

instancemethod sum2[TSupportsAdd, TDefault](__selector, __default)

Parameters
__selector: Callable[[*TSource_co*], *TSupportsAdd*]

__default: *TDefault*

Returns
Union[*TSupportsAdd*, *TDefault*]

Computes the sum of the sequence using the selector. Returns the default value if it is empty.

Example

```
>>> Enumerable([]).sum2(lambda e: int(e * 1000), 880)
880
```

---

instancemethod take(count)

Parameters
count: int

Returns
Enumerable[*TSource_co*]

Returns a specified number of contiguous elements from the start of the sequence.

Example

```
>>> grades = [98, 92, 85, 82, 70, 59, 56]
>>> Enumerable(grades).take(3).to_list()
[98, 92, 85]
```

---

instancemethod take(__index)

Parameters
__index: slice

Returns
Enumerable[*TSource_co*]

Produces a subsequence defined by the given slice notation.

This method always uses a generic list slicing method regardless the implementation of the wrapped iterable.

This method currently is identical to `elements_in()` when it takes a slice.

Example

```
>>> def gen():
...     yield 1; yield 10; yield 100; yield 1000; yield 10000

>>> Enumerable(gen()).take(slice(1, 3)).to_list()
[10, 100]
```

Revisions
v1.0.0: New.

---

instancemethod take_last(count)

Parameters
count: int

Returns
Enumerable[*TSource_co*]

Returns a new sequence that contains the last `count` elements.

Example

```
>>> grades = [98, 92, 85, 82, 70, 59, 56]
>>> Enumerable(grades).take_last(3).to_list()
[70, 59, 56]
```

---

instancemethod take_while(predicate)

Parameters

predicate: Callable[[*TSource_co*], bool]

Returns

Enumerable[*TSource_co*]

Returns elements from the sequence as long as the condition is true and skips the remaining.

Example

```
>>> strs = ['1', '3', '5', '7', '', '1', '4', '5']
>>> Enumerable(strs).take_while(lambda g: g).to_list()
['1', '3', '5', '7']
```

---

instancemethod take_while2(predicate)

Parameters

predicate: Callable[[*TSource_co*, int], bool]

Returns

Enumerable[*TSource_co*]

Returns elements from the sequence as long as the condition is true and skips the remaining. The element's index is used in the predicate function.

---

instancemethod to_dict[TKey, TValue](key_selector, __value_selector)

Parameters

key_selector: Callable[[*TSource_co*], *TKey*]

__value_selector: Callable[[*TSource_co*], *TValue*]

Returns

Dict[*TKey*, *TValue*]

Enumerates all values and returns a dict containing them. key_selector and value_selector are used to select keys and values.

---

instancemethod `to_dict[TKey](key_selector)`

Parameters
key_selector: `Callable[[TSource_co], TKey]`

Returns
`Dict[TKey, TSource_co]`

Enumerates all values and returns a dict containing them. key_selector is used to select keys.

---

instancemethod `to_set()`

Returns
`Set[TSource_co]`

Enumerates all values and returns a set containing them.

---

instancemethod `to_list()`

Returns
`List[TSource_co]`

Enumerates all values and returns a list containing them.

---

instancemethod `to_lookup[TKey, TValue](key_selector, __value_selector)`

Parameters
key_selector: `Callable[[TSource_co], TKey]`

__value_selector: `Callable[[TSource_co], TValue]`

Returns
Lookup`[TKey, TValue]`

Enumerates all values and returns a lookup containing them according to specified key selector and value selector. The values within each group are in the same order as in self.

Example

```
>>> food = [
...     ('main', 'ramen'), ('main', 'noodles'), ('side', 'chicken'),
...     ('main', 'spaghetti'), ('snack', 'popcorns'), ('side', 'apples'),
...     ('side', 'orange'), ('drink', 'coke'), ('main', 'birthdaycake'),
... ]
>>> lookup = Enumerable(food).to_lookup(lambda e: e[0], lambda e: e[1])
>>> lookup.select(lambda grouping: grouping.key).to_list()
['main', 'side', 'snack', 'drink']
>>> if 'side' in lookup:
...     print(lookup['side'].to_list())
['chicken', 'apples', 'orange']
```

Revisions
v0.2.1: Added preliminary support for unhashable keys.

---

instancemethod `to_lookup[TKey](key_selector)`

Parameters
key_selector: `Callable[[TSource_co], TKey]`

Returns
Lookup[*TKey*, *TSource_co*]

Enumerates all values and returns a lookup containing them according to the specified key selector. The values within each group are in the same order as in self.

Revisions
v0.2.1: Added preliminary support for unhashable keys.

---

instancemethod `union(second)`

Parameters
second: `Iterable[TSource_co]`

Returns
Enumerable[*TSource_co*]

Produces the set union of two sequences: self + second.

Example

```
>>> gen = (i for i in range(5))
>>> lst = [5, 3, 9, 7, 5, 9, 3, 7]
>>> Enumerable(gen).union(lst).to_list()
[0, 1, 2, 3, 4, 5, 9, 7]
```

Revisions
v0.2.1: Added preliminary support for unhashable values.

---

instancemethod `union_by(second, key_selector)`

Parameters
second: `Iterable[TSource_co]`

key_selector: `Callable[[TSource_co], object]`

Returns
Enumerable[*TSource_co*]

Produces the set union of two sequences: self + second according to a specified key selector.

Example

```
>>> en = Enumerable([1, 9, -2, -7, 14])
>>> en.union_by([15, 2, -26, -7], abs).to_list()
[1, 9, -2, -7, 14, 15, -26]  # abs(-2) == abs(2)
```

Revisions
v1.0.0: New.

---

instancemethod `where(predicate)`

Parameters
predicate: `Callable[[TSource_co], bool]`

Returns
Enumerable[*TSource_co*]

Filters the sequence of values based on a predicate.

Example

```
>>> strs = ['apple', 'orange', 'Apple', 'xx', 'Grapes']
>>> Enumerable(strs).where(str.istitle).to_list()
['Apple', 'Grapes']
```

---

instancemethod `where2(predicate)`

Parameters
predicate: `Callable[[TSource_co, int], bool]`

Returns
Enumerable[*TSource_co*]

Filters the sequence of values based on a predicate. Each element's index is used in the predicate logic.

Example

```
>>> ints = [0, 30, 20, 15, 90, 85, 40, 75]
>>> Enumerable(ints).where2(lambda e, i: e <= i * 10).to_list()
[0, 20, 15, 40]
```

---

instancemethod `zip[TOther](__second)`

Parameters
__second: `Iterable[TOther]`

Returns
Enumerable[Tuple[*TSource_co*, *TOther*]]

Produces a sequence of 2-element tuples from the two sequences.

Example

```
>>> ints = [1, 2, 3, 4]
>>> dims = ['x', 'y', 'z', 't', 'u', 'v']
>>> Enumerable(ints).zip(dims).to_list()
[(1, 'x'), (2, 'y'), (3, 'z'), (4, 't')]
```

---

instancemethod zip[TOther, TOther2](__second, __third)

Parameters

__second: Iterable[*TOther*]

__third: Iterable[*TOther2*]

Returns

Enumerable[Tuple[*TSource_co*, *TOther*, *TOther2*]]

Revisions

v0.1.1: New.

---

instancemethod zip[TOther, TOther2, TOther3](__second, __third, __fourth)

Parameters

__second: Iterable[*TOther*]

__third: Iterable[*TOther2*]

__fourth: Iterable[*TOther3*]

Returns

Enumerable[Tuple[*TSource_co*, *TOther*, *TOther2*, *TOther3*]]

Revisions

v0.1.1: New.

---

instancemethod zip[TOther, TOther2, TOther3, TOther4](__second, __third, __fourth, __fifth)

Parameters

__second: Iterable[*TOther*]

__third: Iterable[*TOther2*]

__fourth: Iterable[*TOther3*]

__fifth: Iterable[*TOther4*]

Returns

Enumerable[Tuple[*TSource_co*, *TOther*, *TOther2*, *TOther3*, *TOther4*]]

Revisions

v0.1.1: New.

---

instancemethod zip(__second, __third, __fourth, __fifth, __sixth, *iters)

Parameters

__second: Iterable[Any]

__third: Iterable[Any]

__fourth: Iterable[Any]

__fifth: Iterable[Any]

__sixth: Iterable[Any]

**iters*: Iterable[Any]

Returns

Enumerable[Tuple[Any, ...]]

Revisions

v0.1.1: New.

---

instancemethod zip2[TOther, TResult](__second, __result_selector)

Parameters

__second: Iterable[*TOther*]

__result_selector: Callable[[*TSource_co*, *TOther*], *TResult*]

Returns

Enumerable[*TResult*]

Applies a specified function to the corresponding elements of two sequences, producing a sequence of the results.

Example

```
>>> ints = [1, 2, 3, 4]
>>> dims = ['x', 'y', 'z', 't', 'u', 'v']
>>> Enumerable(ints).zip2(dims, lambda i, d: f'{i}.{d}').to_list()
['1.x', '2.y', '3.z', '4.t']
```

---

instancemethod zip2[TOther, TOther2, TResult](__second, __third, __result_selector)

Parameters

__second: Iterable[*TOther*]

__third: Iterable[*TOther2*]

__result_selector: Callable[[*TSource_co*, *TOther*, *TOther2*], *TResult*]

Returns

Enumerable[*TResult*]

Revisions

v0.1.1: New.

---

instancemethod zip2[TOther, TOther2, TOther3, TResult](__second, __third, __fourth, __result_selector)

Parameters

__second: Iterable[*TOther*]

__third: Iterable[*TOther2*]

__fourth: Iterable[*TOther3*]

__result_selector: Callable[[*TSource_co*, *TOther*, *TOther2*, *TOther3*], *TResult*]

Returns

Enumerable[*TResult*]

Revisions

v0.1.1: New.

---

instancemethod zip2[TOther, TOther2, TOther3, TOther4, TResult](__second, __third, __fourth, __fifth, __result_selector)

Parameters

__second: Iterable[*TOther*]

__third: Iterable[*TOther2*]

__fourth: Iterable[*TOther3*]

__fifth: Iterable[*TOther4*]

__result_selector: Callable[[*TSource_co*, *TOther*, *TOther2*, *TOther3*, *TOther4*], *TResult*]

Returns

Enumerable[*TResult*]

Revisions

v0.1.1: New.

---

instancemethod zip2(__second, __third, __fourth, __fifth, __sixth, *iters_and_result_selector)

Parameters

__second: Iterable[Any]

__third: Iterable[Any]

__fourth: Iterable[Any]

__fifth: Iterable[Any]

__sixth: Iterable[Any]

**iters_and_result_selector*: Union[Iterable[Any], Callable[..., Any]]

Returns

Enumerable[Any]

Revisions
v0.1.1: New.

---

instancemethod `elements_in(__index)`

Parameters
__index: `slice`

Returns
Enumerable[*TSource_co*]

Produces a subsequence defined by the given slice notation.

This method always uses a generic list slicing method regardless the implementation of the wrapped iterable.

This method currently is identical to `take()` when it takes a slice.

Example

```
>>> def gen():
...     yield 1; yield 10; yield 100; yield 1000; yield 10000

>>> Enumerable(gen()).elements_in(slice(1, 3)).to_list()
[10, 100]
```

---

instancemethod `elements_in(__start, __stop, __step=1)`

Parameters
__start: `int`

__stop: `int`

__step: `int`

Returns
Enumerable[*TSource_co*]

Produces a subsequence with indices that define a slice.

This method always uses a generic list slicing method regardless the implementation of the wrapped iterable.

Example

```
>>> def gen():
...     yield 1; yield 10; yield 100; yield 1000; yield 10000

>>> Enumerable(gen()).elements_in(1, 3).to_list()
[10, 100]
```

---

instancemethod to_tuple()

Returns
Tuple[*TSource_co*, ...]

Enumerates all values and returns a tuple containing them.

Revisions
v0.1.2: New.

CHAPTER

SEVEN

MODULE TYPES_LINQ.GROUPING

7.1 class Grouping[TValue_co, TKey_co]

```
from types_linq.grouping import Grouping
```

Represents a collection of objects that have a common key.

Users should not construct instances of this class directly. Use `Enumerable.group_by()` instead.

7.1.1 Bases

- *Enumerable*[*TValue_co*]
- Generic[*TKey_co*, *TValue_co*]

7.1.2 Members

instanceproperty key

Returns

TKey_co

Gets the key of the grouping.

CHAPTER

EIGHT

MODULE TYPES_LINQ.LOOKUP

8.1 class Lookup[TKey_co, TValue_co]

```
from types_linq.lookup import Lookup
```

A lookup is a one-to-many dictionary. It maps keys to Enumerable sequences of values.

Users should not construct instances of this class directly. Use `Enumerable.to_lookup()` instead.

8.1.1 Bases

- *Enumerable*[*Grouping*[*TKey_co*, *TValue_co*]]

8.1.2 Members

instanceproperty count

Returns
`int`

Gets the number of key-collection pairs.

---

instancemethod __contains__(value)

Parameters
value: `object`

Returns
`bool`

Tests whether key is in the lookup.

---

instancemethod `__len__()`

Returns
`int`

Gets the number of key-collection pairs.

---

instancemethod `__getitem__(key)`

Parameters
key: *TKey_co*

Returns
Enumerable[*TValue_co*]

Gets the collection of values indexed by the specified key, or empty if no such key exists.

---

instancemethod `apply_result_selector[TResult](result_selector)`

Parameters
result_selector: `Callable[[`*TKey_co*`, `*Enumerable*[*TValue_co*]], *TResult*]

Returns
Enumerable[*TResult*]

Applies a transform function to each key and its associated values, then returns the results.

---

instancemethod `contains(value)`

Parameters
value: `object`

Returns
`bool`

Tests whether key is in the lookup.

CHAPTER

NINE

MODULE TYPES_LINQ.MORE_TYPING

Typing utilities used by methods's declarations across the library. For more details, see typing.

Note: Definitions in this module are for documenting purposes only.

9.1 Constants

9.1.1 TAccumulate

Equals

```
TypeVar('TAccumulate')
```

A generic type parameter.

9.1.2 TAverage_co

Equals

```
TypeVar('TAverage_co', covariant=True)
```

A generic covariant type parameter.

9.1.3 TCollection

Equals

```
TypeVar('TCollection')
```

A generic type parameter.

9.1.4 TDefault

Equals
TypeVar('*TDefault*')

A generic type parameter.

9.1.5 TInner

Equals
TypeVar('*TInner*')

A generic type parameter.

9.1.6 TKey

Equals
TypeVar('*TKey*')

A generic type parameter.

9.1.7 TKey2

Equals
TypeVar('*TKey2*')

A generic type parameter.

9.1.8 TKey_co

Equals
TypeVar('*TKey_co*', covariant=True)

A generic covariant type parameter.

9.1.9 TOther

Equals
TypeVar('*TOther*')

A generic type parameter.

9.1.10 TOther2

Equals
TypeVar('*TOther2*')

A generic type parameter.

9.1.11 TOther3

Equals
TypeVar('*TOther3*')

A generic type parameter.

9.1.12 TOther4

Equals
TypeVar('*TOther4*')

A generic type parameter.

9.1.13 TResult

Equals
TypeVar('*TResult*')

A generic type parameter.

9.1.14 TSelf

Equals
TypeVar('*TSelf*')

A generic type parameter.

9.1.15 TSource

Equals
TypeVar('*TSource*')

A generic type parameter.

9.1.16 TSource_co

Equals
TypeVar('*TSource_co*', covariant=True)

A generic covariant type parameter.

9.1.17 TValue

Equals
TypeVar('*TValue*')

A generic type parameter.

9.1.18 TValue_co

Equals
TypeVar('*TValue_co*', covariant=True)

A generic covariant type parameter.

9.1.19 TSupportsLessThan

Equals
TypeVar('*TSupportsLessThan*', bound=*SupportsLessThan*)

A generic type parameter that represents a type that *SupportsLessThan*.

9.1.20 TSupportsAdd

Equals
TypeVar('*TSupportsAdd*', bound=*SupportsAdd*)

A generic type parameter that represents a type that *SupportsAdd*.

9.2 class SupportsAverage[TAverage_co]

Instances of this protocol supports the averaging operation. that is, if `x` is such an instance, and `N` is an integer, then `(x + x + ...) / N` is allowed, and has the type *TAverage_co*.

9.2.1 Bases

- Protocol[*TAverage_co*]

9.2.2 Members

abstract instancemethod __add__[TSelf](__o)

Constraint
self: *TSelf*

Parameters
__o: *TSelf*

Returns
TSelf

---

abstract instancemethod __truediv__(__o)

Parameters
__o: int

Returns
TAverage_co

9.3 class SupportsLessThan

Instances of this protocol supports the < operation.

Even though they may be unimplemented, the existence of < implies the existence of >, and probably ==, !=, <= and >=.

9.3.1 Bases

- Protocol

9.3.2 Members

abstract instancemethod __lt__(__o)

Parameters
__o: Any

Returns
bool

9.4 class SupportsAdd

Instances of this protocol supports the homogeneous + operation.

9.4.1 Bases

- Protocol

9.4.2 Members

abstract instancemethod __add__[TSelf](__o)

Constraint
self: *TSelf*

Parameters
__o: *TSelf*

Returns
TSelf

CHAPTER

TEN

MODULE TYPES_LINQ.ORDERED_ENUMERABLE

10.1 class OrderedEnumerable[TSource_co, TKey]

```
from types_linq.ordered_enumerable import OrderedEnumerable
```

Represents a sorted Enumerable sequence that is sorted by some key.

Users should not construct instances of this class directly. Use `Enumerable.order_by()` instead.

10.1.1 Bases

- *Enumerable*[*TSource_co*]
- Generic[*TSource_co*, *TKey*]

10.1.2 Members

instancemethod create_ordered_enumerable[TKey2](key_selector, comparer, descending)

Parameters

key_selector: Callable[[*TSource_co*], *TKey2*]

comparer: Optional[Callable[[*TKey2*, *TKey2*], int]]

descending: bool

Returns

OrderedEnumerable[*TSource_co*, *TKey2*]

Performs a subsequent ordering on the elements of the sequence according to a key.

Comparer takes two values and return positive ints when lhs > rhs, negative ints if lhs < rhs, and 0 if they are equal.

Revisions

v0.1.2: Fixed incorrect parameter type of comparer.

instancemethod then_by[TSupportsLessThan](key_selector)

Parameters

key_selector: Callable[[*TSource_co*], *TSupportsLessThan*]

Returns

OrderedEnumerable[*TSource_co*, *TSupportsLessThan*]

Performs a subsequent ordering of the elements in ascending order according to key.

Example

```
>>> class Pet(NamedTuple):
...     name: str
...     age: int

>>> pets = [Pet('Barley', 8), Pet('Boots', 4), Pet('Roman', 5), Pet('Daisy', 4)]
>>> Enumerable(pets).order_by(lambda p: p.age) \
...     .then_by(lambda p: p.name)             \
...     .select(lambda p: p.name)              \
...     .to_list()
['Boots', 'Daisy', 'Roman', 'Barley']
```

---

instancemethod then_by[TKey2](key_selector, __comparer)

Parameters

key_selector: Callable[[*TSource_co*], *TKey2*]

__comparer: Callable[[*TKey2*, *TKey2*], int]

Returns

OrderedEnumerable[*TSource_co*, *TKey2*]

Performs a subsequent ordering of the elements in ascending order by using a specified comparer.

Such comparer takes two values and return positive ints when lhs > rhs, negative ints if lhs < rhs, and 0 if they are equal.

---

instancemethod then_by_descending[TSupportsLessThan](key_selector)

Parameters

key_selector: Callable[[*TSource_co*], *TSupportsLessThan*]

Returns

OrderedEnumerable[*TSource_co*, *TSupportsLessThan*]

Performs a subsequent ordering of the elements in descending order according to key.

---

instancemethod then_by_descending[TKey2](key_selector, __comparer)

Parameters

key_selector: `Callable[[TSource_co], TKey2]`

__comparer: `Callable[[TKey2, TKey2], int]`

Returns

`OrderedEnumerable[TSource_co, TKey2]`

Performs a subsequent ordering of the elements in descending order by using a specified comparer.

Such comparer takes two values and return positive ints when lhs > rhs, negative ints if lhs < rhs, and 0 if they are equal.

CHAPTER

ELEVEN

MODULE TYPES_LINQ.TYPES_LINQ_ERROR

11.1 class TypesLinqError

```
from types_linq import TypesLinqError
```

Types-linq has run into problems.

11.1.1 Bases

- Exception

11.2 class InvalidOperationError

```
from types_linq import InvalidOperationError
```

Exception raised when a call is invalid for the object's current state.

11.2.1 Bases

- *TypesLinqError*
- ValueError

11.3 class IndexOutOfRangeError

```
from types_linq import IndexOutOfRangeError
```

An IndexError with types-linq flavour.

11.3.1 Bases

- *TypesLinqError*
- IndexError

CHAPTER

TWELVE

MODULE TYPES_LINQ.MORE.EXTREMA_ENUMERABLE

12.1 class ExtremaEnumerable[TSource_co, TKey]

```
from types_linq.more.extrema_enumerable import ExtremaEnumerable
```

Specialization for manipulating extrema.

Users should not construct instances of this class directly. Use `MoreEnumerable.maxima_by()` instead.

Revisions
v0.2.0: New.

12.1.1 Bases

- *MoreEnumerable*[*TSource_co*]
- Generic[*TSource_co*, *TKey*]

12.1.2 Members

instancemethod take(count)

Parameters
count: `int`

Returns
MoreEnumerable[*TSource_co*]

Returns a specified number of contiguous elements from the start of the sequence.

instancemethod take(__index)

Parameters
__index: `slice`

Returns
Enumerable[*TSource_co*]

Identical to parent.

Revisions
v1.1.0: Fixed incorrect override of `Enumerable.take()` when it takes a slice.

---

instancemethod take_last(count)

Parameters
count: `int`

Returns
MoreEnumerable[*TSource_co*]

Returns a new sequence that contains the last `count` elements.

CHAPTER

THIRTEEN

MODULE TYPES_LINQ.MORE.MORE_ENUMERABLE

13.1 class MoreEnumerable[TSource_co]

```
from types_linq.more import MoreEnumerable
```

MoreEnumerable provides more query methods. Instances of this class can be created by directly constructing, using `as_more()`, or invoking MoreEnumerable methods that return MoreEnumerable instead of Enumerable.

These APIs may have breaking changes more frequently than those in Enumerable class because updates in .NET are happening and sometimes ones of these APIs could be moved to Enumerable with modification, or changed to accommodate changes to Enumerable.

Revisions
v0.2.0: New.

13.1.1 Bases

- *Enumerable*[*TSource_co*]

13.1.2 Members

instancemethod aggregate_right[TAccumulate, TResult](__seed, __func, __result_selector)

Parameters
__seed: *TAccumulate*

__func: Callable[[*TSource_co*, *TAccumulate*], *TAccumulate*]

__result_selector: Callable[[*TAccumulate*], *TResult*]

Returns
TResult

Applies a right-associative accumulator function over the sequence. The seed is used as the initial accumulator value, and the result_selector is used to select the result value.

Revisions
v1.2.0: Fixed annotation for __func.

instancemethod aggregate_right[TAccumulate](__seed, __func)

Parameters

__seed: *TAccumulate*

__func: Callable[[*TSource_co*, *TAccumulate*], *TAccumulate*]

Returns

TAccumulate

Applies a right-associative accumulator function over the sequence. The seed is used as the initial accumulator value.

Example

```
>>> values = [9, 4, 2]
>>> MoreEnumerable(values).aggregate_right('null', lambda e, rr: f'(cons {e} {rr})')
'(cons 9 (cons 4 (cons 2 null)))'
```

Revisions

v1.2.0: Fixed annotation for __func.

---

instancemethod aggregate_right(__func)

Parameters

__func: Callable[[*TSource_co*, *TSource_co*], *TSource_co*]

Returns

TSource_co

Applies a right-associative accumulator function over the sequence. Raises *InvalidOperationError* if there is no value in the sequence.

Example

```
>>> values = ['9', '4', '2', '5']
>>> MoreEnumerable(values).aggregate_right(lambda e, rr: f'({e}+{rr})')
'(9+(4+(2+5)))'
```

Revisions

v1.2.0: Fixed annotation for __func.

---

instancemethod as_more()

Returns

MoreEnumerable[*TSource_co*]

Returns the original MoreEnumerable reference.

---

instancemethod consume()

Returns
None

Consumes the sequence completely. This method iterates the sequence immediately and does not save any intermediate data.

Revisions
v1.1.0: New.

---

instancemethod cycle(count=None)

Parameters
count: Optional[int]

Returns
MoreEnumerable[*TSource_co*]

Repeats the sequence count times.

If count is None, the sequence is infinite. Raises *InvalidOperationError* if count is negative.

Example

```
>>> MoreEnumerable([1, 2, 3]).cycle(3).to_list()
[1, 2, 3, 1, 2, 3, 1, 2, 3]
```

Revisions
v1.1.0: New.

---

instancemethod enumerate(start=0)

Parameters
start: int

Returns
MoreEnumerable[Tuple[int, *TSource_co*]]

Returns a sequence of tuples containing the index and the value from the source sequence. start is used to specify the starting index.

Example

```
>>> ints = [2, 4, 6]
>>> MoreEnumerable(ints).enumerate().to_list()
[(0, 2), (1, 4), (2, 6)]
```

Revisions
v1.0.0: New.

---

instancemethod except_by2(second, key_selector)

Parameters

second: `Iterable[TSource_co]`

key_selector: `Callable[[TSource_co], object]`

Returns

MoreEnumerable[*TSource_co*]

Produces the set difference of two sequences: self - second, according to a key selector that determines "distinctness". Note the second iterable is homogenous to self.

Example

```
>>> first = [(16, 'x'), (9, 'y'), (12, 'd'), (16, 't')]
>>> second = [(24, 'd'), (77, 'y')]
>>> MoreEnumerable(first).except_by2(second, lambda x: x[1]).to_list()
[(16, 'x'), (16, 't')]
```

Revisions

v1.0.0: Renamed from `except_by()` to this name to accommodate an update to Enumerable class.

v0.2.1: Added preliminary support for unhashable keys.

---

instancemethod flatten()

Returns

MoreEnumerable[Any]

Flattens the sequence containing arbitrarily-nested subsequences.

Note: the nested objects must be Iterable to be flatten. Instances of `str` or `bytes` are not flattened.

Example

```
>>> lst = ['apple', ['orange', ['juice', 'mango'], 'delta function']]
>>> MoreEnumerable(lst).flatten().to_list()
['apple', 'orange', 'juice', 'mango', 'delta function']
```

---

instancemethod flatten(__predicate)

Parameters

__predicate: `Callable[[Iterable[Any]], bool]`

Returns

MoreEnumerable[Any]

Flattens the sequence containing arbitrarily-nested subsequences. A predicate function determines whether a nested iterable should be flattened or not.

Note: the nested objects must be Iterable to be flatten.

---

instancemethod `flatten2(selector)`

Parameters
selector: `Callable[[Any], Optional[Iterable[object]]]`

Returns
MoreEnumerable`[Any]`

Flattens the sequence containing arbitrarily-nested subsequences. A selector is used to select a subsequence based on the object's properties. If the selector returns None, then the object is considered a leaf.

---

instancemethod `for_each(action)`

Parameters
action: `Callable[[`*TSource_co*`], object]`

Returns
`None`

Executes the given function on each element in the source sequence. The return values are discarded.

Example

```
>>> def gen():
...     yield 116; yield 35; yield -9

>>> Enumerable(gen()).where(lambda x: x > 0).as_more().for_each(print)
116
35
```

---

instancemethod `for_each2(action)`

Parameters
action: `Callable[[`*TSource_co*`, int], object]`

Returns
`None`

Executes the given function on each element in the source sequence. Each element's index is used in the logic of the function. The return values are discarded.

---

instancemethod `interleave(*iters)`

Parameters
**iters*: `Iterable[`*TSource_co*`]`

Returns
MoreEnumerable`[`*TSource_co*`]`

Interleaves the elements of two or more sequences into a single sequence, skipping sequences if they are consumed.

Example

```
>>> MoreEnumerable(['1', '2']).interleave(['4', '5', '6'], ['7', '8', '9']).to_
↪list()
['1', '4', '7', '2', '5', '8', '6', '9']
```

---

instancemethod maxima_by[TSupportsLessThan](selector)

Parameters

selector: Callable[[*TSource_co*], *TSupportsLessThan*]

Returns

ExtremaEnumerable[*TSource_co*, *TSupportsLessThan*]

Returns the maximal elements of the sequence based on the given selector.

Example

```
>>> strings = ['foo', 'bar', 'cheese', 'orange', 'baz', 'spam', 'egg', 'toasts',
↪'dish']
>>> MoreEnumerable(strings).maxima_by(len).to_list()
['cheese', 'orange', 'toasts']
>>> MoreEnumerable(strings).maxima_by(lambda x: x.count('e')).first()
'cheese'
```

---

instancemethod maxima_by[TKey](selector, __comparer)

Parameters

selector: Callable[[*TSource_co*], *TKey*]

__comparer: Callable[[*TKey*, *TKey*], int]

Returns

ExtremaEnumerable[*TSource_co*, *TKey*]

Returns the maximal elements of the sequence based on the given selector and the comparer.

Such comparer takes two values and return positive ints when lhs > rhs, negative ints if lhs < rhs, and 0 if they are equal.

---

instancemethod minima_by[TSupportsLessThan](selector)

Parameters

selector: Callable[[*TSource_co*], *TSupportsLessThan*]

Returns

ExtremaEnumerable[*TSource_co*, *TSupportsLessThan*]

Returns the minimal elements of the sequence based on the given selector.

---

instancemethod minima_by[TKey](selector, __comparer)

Parameters

selector: Callable[[*TSource_co*], *TKey*]

__comparer: Callable[[*TKey*, *TKey*], int]

Returns

ExtremaEnumerable[*TSource_co*, *TKey*]

Returns the minimal elements of the sequence based on the given selector and the comparer.

Such comparer takes two values and return positive ints when lhs > rhs, negative ints if lhs < rhs, and 0 if they are equal.

---

instancemethod pipe(action)

Parameters

action: Callable[[*TSource_co*], object]

Returns

MoreEnumerable[*TSource_co*]

Executes the given action on each element in the sequence and yields it. Return values of action are discarded.

Example

```
>>> store = set()
>>> MoreEnumerable([1, 2, 2, 1]).pipe(store.add).where(lambda x: x % 2 == 0).to_
↪list()
[2, 2]
>>> store
{1, 2}
```

Revisions

v0.2.1: New.

---

instancemethod pre_scan[TAccumulate](identity, transformation)

Parameters

identity: *TAccumulate*

transformation: Callable[[*TAccumulate*, *TSource_co*], *TAccumulate*]

Returns

MoreEnumerable[*TAccumulate*]

Performs a pre-scan (exclusive prefix sum) over the sequence. Such scan returns an equal-length sequence where the first element is the identity, and i-th element (i>1) is the sum of the first i-1 (and identity) elements in the original sequence.

Example

```
>>> values = [9, 4, 2, 5, 7]
>>> MoreEnumerable(values).pre_scan(0, lambda acc, e: acc + e).to_list()
[0, 9, 13, 15, 20]
>>> MoreEnumerable([]).pre_scan(0, lambda acc, e: acc + e).to_list()
[]
```

Revisions

v1.2.0: New.

---

instancemethod rank[TSupportsLessThan](*, method=RankMethods.dense)

Constraint

self: *MoreEnumerable*[*TSupportsLessThan*]

Parameters

method: *RankMethods*

Returns

MoreEnumerable[int]

Ranks each item in the sequence in descending order using the method provided.

Example

```
>>> scores = [1, 4, 77, 23, 23, 4, 9, 0, -7, 101, 23]
>>> MoreEnumerable(scores).rank().to_list()
[6, 5, 2, 3, 3, 5, 4, 7, 8, 1, 3]  # 101 is largest, so has rank of 1

>>> MoreEnumerable(scores).rank(method=RankMethods.competitive).to_list()
[9, 7, 2, 3, 3, 7, 6, 10, 11, 1, 3]  # there are no 4th or 5th since there
                                     # are three 3rd's

>>> MoreEnumerable(scores).rank(method=RankMethods.ordinal).to_list()
[9, 7, 2, 3, 4, 8, 6, 10, 11, 1, 5]  # as in sorting
```

Revisions

v1.2.1: Added method parameter to support more ranking methods.

v1.0.0: New.

---

instancemethod rank(__comparer, *, method=RankMethods.dense)

Parameters

__comparer: Callable[[*TSource_co*, *TSource_co*], int]

method: *RankMethods*

Returns

MoreEnumerable[int]

Ranks each item in the sequence in descending order using the given comparer and the method.

Such comparer takes two values and return positive ints when lhs > rhs, negative ints if lhs < rhs, and 0 if they are equal.

Revisions

v1.2.1: Added method parameter to support more ranking methods.

v1.0.0: New.

---

instancemethod `rank_by[TSupportsLessThan](key_selector, *, method=RankMethods.dense)`

Parameters

key_selector: `Callable[[TSource_co], TSupportsLessThan]`

method: *RankMethods*

Returns

MoreEnumerable[int]

Ranks each item in the sequence in descending order using the given selector and the method.

Example

```
>>> scores = [
...     {'name': 'Frank', 'score': 75},
...     {'name': 'Alica', 'score': 90},
...     {'name': 'Erika', 'score': 99},
...     {'name': 'Rogers', 'score': 90},
... ]

>>> MoreEnumerable(scores).rank_by(lambda x: x['score']) \
...     .zip(scores) \
...     .group_by(lambda t: t[0], lambda t: t[1]['name']) \
...     .to_dict(lambda g: g.key, lambda g: g.to_list())
{3: ['Frank'], 2: ['Alica', 'Rogers'], 1: ['Erika']}
```

Revisions

v1.2.1: Added method parameter to support more ranking methods.

v1.0.0: New.

---

instancemethod `rank_by[TKey](key_selector, __comparer, *, method=RankMethods.dense)`

Parameters

key_selector: `Callable[[TSource_co], TKey]`

__comparer: `Callable[[TKey, TKey], int]`

method: *RankMethods*

Returns

MoreEnumerable[int]

Ranks each item in the sequence in descending order using the given selector, comparer and the method.

Such comparer takes two values and return positive ints when lhs > rhs, negative ints if lhs < rhs, and 0 if they are equal.

Revisions

v1.2.1: Added method parameter to support more ranking methods.

v1.0.0: New.

---

instancemethod run_length_encode()

Returns

MoreEnumerable[Tuple[*TSource_co*, int]]

Run-length encodes the sequence into a sequence of tuples where each tuple contains an (the first) element and its number of contingent occurrences, where equality is based on ==.

Example

```
>>> MoreEnumerable('abbcaeeeaa').run_length_encode().to_list()
[('a', 1), ('b', 2), ('c', 1), ('a', 1), ('e', 3), ('a', 2)]
```

Revisions

v1.1.0: New.

---

instancemethod run_length_encode(__comparer)

Parameters

__comparer: Callable[[*TSource_co*, *TSource_co*], bool]

Returns

MoreEnumerable[Tuple[*TSource_co*, int]]

Run-length encodes the sequence into a sequence of tuples where each tuple contains an (the first) element and its number of contingent occurrences, where equality is determined by the comparer.

Example

```
>>> MoreEnumerable('abBBbcaEeeff') \
>>>     .run_length_encode(lambda x, y: x.lower() == y.lower()).to_list()
[('a', 1), ('b', 4), ('c', 1), ('a', 1), ('E', 3), ('f', 2)]
```

Revisions

v1.1.0: New.

---

instancemethod scan(__transformation)

Parameters

__transformation: Callable[[*TSource_co*, *TSource_co*], *TSource_co*]

Returns

MoreEnumerable[*TSource_co*]

Performs a inclusive prefix sum over the sequence. Such scan returns an equal-length sequence where the i-th element is the sum of the first i elements in the original sequence.

Example

```
>>> values = [9, 4, 2, 5, 7]
>>> MoreEnumerable(values).scan(lambda acc, e: acc + e).to_list()
[9, 13, 15, 20, 27]
>>> MoreEnumerable([]).scan(lambda acc, e: acc + e).to_list()
[]
```

Example

```
>>> # running max
>>> fruits = ['apple', 'mango', 'orange', 'passionfruit', 'grape']
>>> MoreEnumerable(fruits).scan(lambda acc, e: e if len(e) > len(acc) else acc).to_
↪list()
['apple', 'apple', 'orange', 'passionfruit', 'passionfruit']
```

Revisions

v1.2.0: New.

---

instancemethod scan[TAccumulate](__seed, __transformation)

Parameters

__seed: *TAccumulate*

__transformation: Callable[[*TAccumulate*, *TSource_co*], *TAccumulate*]

Returns

MoreEnumerable[*TAccumulate*]

Like Enumerable.aggregate(seed, transformation) except that the intermediate results are included in the result sequence.

Example

```
>>> Enumerable.range(1, 5).as_more().scan(-1, lambda acc, e: acc * e).to_list()
[-1, -1, -2, -6, -24, -120]
```

Revisions

v1.2.0: New.

---

instancemethod scan_right(__func)

Parameters

__func: Callable[[*TSource_co*, *TSource_co*], *TSource_co*]

Returns

MoreEnumerable[*TSource_co*]

Performs a right-associative inclusive prefix sum over the sequence. This is the right-associative version of MoreEnumerable.scan(func).

Example

```
>>> values = ['9', '4', '2', '5']
>>> MoreEnumerable(values).scan_right(lambda e, rr: f'({e}+{rr})').to_list()
['(9+(4+(2+5)))', '(4+(2+5))', '(2+5)', '5']
>>> MoreEnumerable([]).scan_right(lambda e, rr: e + rr).to_list()
[]
```

Revisions

v1.2.0: New.

---

instancemethod scan_right[TAccumulate](__seed, __func)

Parameters

__seed: *TAccumulate*

__func: Callable[[*TSource_co*, *TAccumulate*], *TAccumulate*]

Returns

MoreEnumerable[*TAccumulate*]

The right-associative version of MoreEnumerable.scan(seed, func).

Example

```
>>> values = [9, 4, 2]
>>> MoreEnumerable(values).scan_right('null', lambda e, rr: f'(cons {e} {rr})').to_
↪list()
['(cons 9 (cons 4 (cons 2 null)))', '(cons 4 (cons 2 null))', '(cons 2 null)', 'null
↪']
```

Revisions

v1.2.0: New.

---

instancemethod segment(new_segment_predicate)

Parameters

new_segment_predicate: Callable[[*TSource_co*], bool]

Returns

MoreEnumerable[*MoreEnumerable*[*TSource_co*]]

Splits the sequence into segments by using a detector function that returns True to signal a new segment.

Example

```
>>> values = [0, 1, 2, 4, -4, -2, 6, 2, -2]
>>> MoreEnumerable(values).segment(lambda x: x < 0).select(lambda x: x.to_list()).
↪to_list()
[[0, 1, 2, 4], [-4], [-2, 6, 2], [-2]]
```

Revisions

v1.2.0: New.

---

instancemethod segment2(new_segment_predicate)

Parameters
new_segment_predicate: Callable[[*TSource_co*, int], bool]

Returns
MoreEnumerable[*MoreEnumerable*[*TSource_co*]]

Splits the sequence into segments by using a detector function that returns True to signal a new segment. The element's index is used in the detector function.

Example

```
>>> values = [0, 1, 2, 4, -4, -2, 6, 2, -2]
>>> MoreEnumerable(values).segment2(lambda x, i: x < 0 or i % 3 == 0) \
...     .select(lambda x: x.to_list()) \
...     .to_list()
[[0, 1, 2], [4], [-4], [-2], [6, 2], [-2]]
```

Revisions
v1.2.0: New.

---

instancemethod segment3(new_segment_predicate)

Parameters
new_segment_predicate: Callable[[*TSource_co*, *TSource_co*, int], bool]

Returns
MoreEnumerable[*MoreEnumerable*[*TSource_co*]]

Splits the sequence into segments by using a detector function that returns True to signal a new segment. The last element and the current element's index are used in the detector function.

Example

```
>>> values = [0, 1, 2, 4, -4, -2, 6, 2, -2]
>>> MoreEnumerable(values).segment3(lambda curr, prev, i: curr * prev < 0) \
...     .select(lambda x: x.to_list()) \
...     .to_list()
[[0, 1, 2, 4], [-4, -2], [6, 2], [-2]]
```

Revisions
v1.2.0: New.

---

staticmethod traverse_breath_first[TSource](root, children_selector)

Parameters

root: *TSource*

children_selector: Callable[[*TSource*], Iterable[*TSource*]]

Returns

MoreEnumerable[*TSource*]

Traverses the tree (graph) from the root node in a breath-first fashion. A selector is used to select children of each node.

Graphs are not checked for cycles or duplicates visits. If the resulting sequence needs to be finite then it is the responsibility of children_selector to ensure that duplicate nodes are not visited.

Example

```
>>> tree = { 3: [1, 4], 1: [0, 2], 4: [5] }
>>> MoreEnumerable.traverse_breath_first(3, lambda x: tree.get(x, [])) \
>>>     .to_list()
[3, 1, 4, 0, 2, 5]
```

---

staticmethod traverse_depth_first[TSource](root, children_selector)

Parameters

root: *TSource*

children_selector: Callable[[*TSource*], Iterable[*TSource*]]

Returns

MoreEnumerable[*TSource*]

Traverses the tree (graph) from the root node in a depth-first fashion. A selector is used to select children of each node.

Graphs are not checked for cycles or duplicates visits. If the resulting sequence needs to be finite then it is the responsibility of children_selector to ensure that duplicate nodes are not visited.

Example

```
>>> tree = { 3: [1, 4], 1: [0, 2], 4: [5] }
>>> MoreEnumerable.traverse_depth_first(3, lambda x: tree.get(x, [])) \
>>>     .to_list()
[3, 1, 0, 2, 4, 5]
```

---

instancemethod traverse_topological(children_selector)

Parameters

children_selector: Callable[[*TSource_co*], Iterable[*TSource_co*]]

Returns

MoreEnumerable[*TSource_co*]

Traverses the graph in topological order, A selector is used to select children of each node. The ordering created from this method is a variant of depth-first traversal and ensures duplicate nodes are output once.

To invoke this method, the self sequence contains nodes with zero in-degrees to start the iteration. Passing a list of all nodes is allowed although not required.

Raises *DirectedGraphNotAcyclicError* if the directed graph contains a cycle and the topological ordering cannot be produced.

Example

```
>>> adj = { 5: [2, 0], 4: [0, 1], 2: [3], 3: [1] }
>>> MoreEnumerable([5, 4]).traverse_topological(lambda x: adj.get(x, [])) \
>>>     .to_list()
[5, 2, 3, 4, 0, 1]
```

Revisions

v1.2.1: New.

---

instancemethod traverse_topological2(children_selector, key_selector)

Parameters

children_selector: `Callable[[TSource_co], Iterable[TSource_co]]`

key_selector: `Callable[[TSource_co], object]`

Returns

MoreEnumerable[*TSource_co*]

Traverses the graph in topological order, A selector is used to select children of each node. The ordering created from this method is a variant of depth-first traversal and ensures duplicate nodes are output once. A key selector is used to determine equality between nodes.

To invoke this method, the self sequence contains nodes with zero in-degrees to start the iteration. Passing a list of all nodes is allowed although not required.

Raises *DirectedGraphNotAcyclicError* if the directed graph contains a cycle and the topological ordering cannot be produced.

Revisions

v1.2.1: New.

CHAPTER

FOURTEEN

MODULE TYPES_LINQ.MORE.MORE_ENUMS

14.1 class RankMethods

```
from types_linq.more import RankMethods
```

Enumeration to select different methods of assigning rankings when breaking ties.

Revisions
v1.2.1: New.

14.1.1 Bases

- Enum

14.1.2 Fields

dense

Equals
auto()

Items that compare equally receive the same ranking, and the next items get the immediately following ranking. *(1223)*

---

competitive

Equals
auto()

Items that compare equally receive the same highest ranking, and gaps are left out. *(1224)*

---

ordinal

Equals
`auto()`

Each item receives unique rankings. *(1234)*

CHAPTER

FIFTEEN

MODULE TYPES_LINQ.MORE.MORE_ERROR

15.1 class DirectedGraphNotAcyclicError

```
from types_linq.more import DirectedGraphNotAcyclicError
```

Exception raised when a cycle exists in a graph.

Revisions
v1.2.1: New.

15.1.1 Bases

- *InvalidOperationError*

15.1.2 Members

instanceproperty cycle

Returns
`Tuple[object, object]`

The two elements (A, B) in this tuple are part of a cycle. There exists an edge from A to B, and a path from B back to A. A and B may be identical.

Example

```
>>> adj = { 5: [2, 0], 4: [0, 1], 2: [3], 3: [1, 5] }
>>> try:
>>>     MoreEnumerable([5, 4]).traverse_topological(lambda x: adj.get(x, [])) \
>>>         .consume()
>>> except DirectedGraphNotAcyclicError as e:
>>>     print(e.cycle)
(3, 5)  # 3 -> 5 -> 2 -> 3
```