carriage Documentation

Release 0.1.6

Yen, Tzu-Hsi

Aug 26, 2018

Contents

carriage aims to make your Python coding life easier. It includes a bunch of powerful collection classes with many practical methods you might use everyday. You can write your code faster, make your code more readable, and test your data pipeline more less painfully.

`carriage` is a Python package hosted on PyPI and works only on Python 3.6 up.

Just like other Python package, install it by pip into a virtualenv, or use pipenev to automatically create and manage the virtualenv.

```
$ pip install carriage
```

CHAPTER 1

Getting Start

All collection classes can be imported from the top level of this package.

```
from carriage import Row, Stream, StreamTable, X, Xcall, Map
from carriage import Array, Optional, Some, Nothing
```

1.1 Row

Row is a handy and more powerful namedtuple. You can create arbitrary *Row* anytime without declaring fields in advance.

```
>>> row = Row(x=3, y=4)
>>> row.x
3
>>> row2 = row.evolve(y=6, z=5)
>>> row3 = row2.without('y')
>>> row
Row(x=3, y=4)
>>> row2
Row(x=3, y=6, z=5)
>>> row3
Row(x=3, z=5)
```

1.2 Stream

Stream is a very powerful wrapper type for any iterable object. You can write less code to transform, inspect, and manipulate any iterable. And with the property of lazy-evaluating, building and testing the pipeline for handling big, long sequential data are now faster, easier and painlessly.

```
>>> Stream(range(5, 8)).map(X * 2).take(2).to_list()
[10, 12]
```

1.3 StreamTable

StreamTable is a subclass of Stream but it assumes all elements are in Row type. This requirement allows StreamTable to provide a more refined interface.

```
>>> stb = StreamTable.from_tuples(
...         [('joe', 170, 59), ('joy', 160, 54), ('may', 163, 55)],
...         fields=('name', 'height', 'weight'))
>>> stb.show()
| name    |   height |   weight |
|---------+----------+----------|
| joe     |      170 |       59 |
| joy     |      160 |       54 |
| may     |      163 |       55 |
>>> stb_bmi = stb.select('name', bmi=X.weight / (X.height/100)**2)
>>> stb_bmi.show()
| name    |      bmi |
|---------+----------|
| joe     |  20.4152 |
| joy     |  21.0937 |
| may     |  20.7008 |
>>> stb_bmi.where(X.bmi > 20.5).show()
| name    |      bmi |
|---------+----------|
| joy     |  21.0937 |
| may     |  20.7008 |
```

1.4 X, Xcall

X and *Xcall* are function creators. Make your lambda function more readable and elegant. See examples above in Stream and StreamTable sections.

1.5 Map, Array

Map and *Array* work just like Python primitive `dict` and `list` but enhanced with a lot of practical methods.

```
>>> Map(joe=32, may=59, joy=31).remove('joy')
Map({'joe': 32, 'may': 59})
>>> Array([1, 2, 3, 4, 5]).filter(lambda n: n % 2 == 0)
Array([2, 4])
```

1.6 Optional

Optional is a object wrapper for handling errors. It makes `None` value, exceptions or other unexpected condition won't break your data processing pipeline.

Read the following API References for further information of each collection class.

CHAPTER 2

API References

2.1 Row: Better named tuple for everyday use

class carriage.**Row**

A named tuple like type without the need of declaring field names in advance.

A Row object can be created anytime when you need it.

```
>>> Row(name='Joe', age=30, height=170)
Row(name='Joe', age=30, height=170)
```

```
>>> Row.from_values([1, 2, 3], fields=['x', 'y', 'z'])
Row(x=1, y=2, z=3)
```

If you are too lazy to name the fields.

```
>>> Row.from_values([1, 'a', 9])
Row(f0=1, f1='a', f2=9)
```

You can access field using index or field name in `O(1)`.

```
>>> row = Row(name='Joe', age=30, height=170)
>>> row.name
'Joe'
>>> row[2]
170
```

And it provides some useful method for transforming, converting. Because Row is immutable type, all these method create a new Row object.

```
>>> row.evolve(height=180)  # I hope so
Row(name='Joe', age=30, height=180)
```

```
>>> row.evolve(age=row.age + 1)
Row(name='Joe', age=31, height=170)
```

```
>>> row.to_dict()
{'name': 'Joe', 'age': 30, 'height': 170}
```

```
>>> row.to_map()
Map({'name': 'Joe', 'age': 30, 'height': 170})
```

Row is iterable. You can unpack it.

```
>>> name, age, height = row
>>> name
'Joe'
>>> age
30
```

evolve (***kwargs*)
Create a new Row by replacing or adding other fields

```
>>> row = Row(x=23, y=9)
>>> row.evolve(y=12)
Row(x=23, y=12)
>>> row.evolve(z=3)
Row(x=23, y=9, z=3)
```

classmethod from_dict (*adict*, *fields=None*)
Create Row from a iterable

```
>>> Row.from_dict({'name': 'Joe', 'age': 30})
Row(name='Joe', age=30)
```

classmethod from_values (*values*, *fields=None*)
Create Row from values

```
>>> Row.from_values([1, 2, 3])
Row(f0=1, f1=2, f2=3)
>>> Row.from_values([1, 2, 3], fields=['x', 'y', 'z'])
Row(x=1, y=2, z=3)
```

get (*field*, *fillvalue=None*)
Get field

```
>>> Row(x=3, y=4).get('x')
3
>>> Row(x=3, y=4).get('z', 0)
0
```

get_opt (*field*)
Get field in Optional type

```
>>> from carriage.optional import Some, Nothing
>>> Row(x=3, y=4).get_opt('x')
Some(3)
>>> Row(x=3, y=4).get_opt('z')
Nothing
```

Parameters **field** (*str*) – field name

Returns

- *Just(value) if field exist*
- *Nothing if field doesn't exist*

has_field(*field*)
Has field

```
>>> Row(x=3, y=4).has_field('x')
True
```

iter_fields()
Convert to rows

```
>>> list(Row(x=3, y=4).iter_fields())
[Row(field='x', value=3), Row(field='y', value=4)]
```

merge(**rows*)
Create a new merged Row. If there's duplicated field name, keep the last value.

```
>>> row = Row(x=2, y=3)
>>> row.merge(Row(y=4, z=5), Row(z=6, u=7))
Row(x=2, y=4, z=6, u=7)
```

project(**fields*)
Create a new Row by keeping only specified fields

```
>>> row = Row(x=2, y=3, z=4)
>>> row.project('x', 'y')
Row(x=2, y=3)
```

rename_fields(***kwargs*)
Create a new Row that field names renamed.

```
>>> row = Row(a=2, b=3, c=4)
>>> row.rename_fields(a='x', b='y')
Row(x=2, y=3, c=4)
```

to_dict()
Convert to dict

to_fields()
Convert to rows

```
>>> Row(x=3, y=4).to_fields()
[Row(field='x', value=3), Row(field='y', value=4)]
```

to_list()
Convert to list

to_map()
Convert to Map

to_tuple()
Convert to tuple

without(**fields*)
Create a new Row by removing only specified fields

```
>>> row = Row(x=2, y=3, z=4)
>>> row.without('z')
Row(x=2, y=3)
```

2.2 `Stream`: Lazy-evaluating sequential collection type

class carriage.**Stream**(*iterable*, ***, *pipeline=None*)
An iterable wrapper for building a lazy-evaluating sequence transformation pipeline.

Stream is initiated by providing any iterable object like list, tuple, iterator and even an infinite one.

```
>>> strm = Stream(range(10))
>>> strm = Stream([1, 2, 3])
```

Some classmethods are provided for creating common Stream instances.

```
>>> strm = Stream.range(0, 10, 2)
>>> strm = Stream.count(0, 5)
```

Stream instance is immutable. Calling a transforamtion function would create a new Stream instance everytime. But don't worry, because of it's lazy-evaluating characteristic, no duplicated data are generated.

```
>>> strm1 = Stream.range(5, 10)
>>> strm2 = strm1.map(lambda n: n * 2)
>>> strm3 = strm1.map(lambda n: n * 3)
>>> strm1 is strm2 or strm1 is strm3 or strm2 is strm3
False
```

To evaluate a Stream instance, call an action function.

```
>>> strm = Stream.range(5, 10).map(lambda n: n * 2).take(3)
>>> strm.sum()
36
>>> strm.to_list()
[10, 12, 14]
```

accumulate(*func=None*)
Create a new Stream of calling `itertools.accumulate`

appended(*elem*)
Create a new Stream that extends source Stream with another element.

cache()
Cache result

chunk(*n*, *strict=False*)
divide elements into chunks of n elements

```
>>> s = Stream.range(5)
>>> s.chunk(2).to_list()
[Row(f0=0, f1=1), Row(f0=2, f1=3), Row(f0=4)]
>>> s.chunk(2, strict=True).to_list()
[Row(f0=0, f1=1), Row(f0=2, f1=3)]
```

classmethod count(*start*, *step=1*)
Create a infinite consecutive Stream

```
>>> Stream.count(0, 3).take(3).to_list()
[0, 3, 6]
```

classmethod cycle(*iterable*)
Create a Stream cycling a iterable

```
>>> Stream.cycle([1,2]).take(5).to_list()
[1, 2, 1, 2, 1]
```

dict_as_row(*fields=None*)
Create a new Stream with elements as Row objects

```
>>> stm = Stream([{'name': 'John', 'age': 35},
...               {'name': 'Frank', 'age': 28}])
>>> stm.dict_as_row().to_list()
[Row(name='John', age=35), Row(name='Frank', age=28)]
>>> stm.dict_as_row(['age', 'name']).to_list()
[Row(age=35, name='John'), Row(age=28, name='Frank')]
```

distincted(*key_func=None*)
Create a new Stream with non-repeating elements. And elements are with the same order of first occurence in the source Stream.

```
>>> Stream.range(10).distincted(lambda n: n//3).to_list()
[0, 3, 6, 9]
```

drop(*n*)
Create a new Stream with first n element dropped

```
>>> Stream(dict(a=3, b=4, c=5).items()).drop(2).to_list()
[('c', 5)]
```

drop_while(*pred*)
Create a new Stream without elements as long as predicate evaluates to true.

dropwhile(*pred*)
Create a new Stream without elements as long as predicate evaluates to true.

extended(*iterable*)
Create a new Stream that extends source Stream with another iterable

filter(*pred*)
Create a new Stream contains only elements passing predicate

```
>>> Stream.range(10).filter(lambda n: n % 2 == 0).to_list()
[0, 2, 4, 6, 8]
```

filter_false(*pred*)
Create a new Stream contains only elements not passing predicate

```
>>> Stream.range(10).filter_false(lambda n: n % 2 == 0).to_list()
[1, 3, 5, 7, 9]
```

find(*pred*)
Get first element satifying predicate

```
>>> Stream.range(5, 100).find(lambda n: n % 7 == 0)
7
```

Returns

Return type element

find_opt(*pred*)
Optionally get first element satifying predicate. Return Some(element) if exist Otherwise return Nothing

```
>>> Stream.range(5, 100).find_opt(lambda n: n * 3 + 5 == 40)
Nothing
>>> Stream.range(5, 100).find_opt(lambda n: n % 7 == 0)
Some(7)
```

Returns

Return type *Optional*[element]

first()
Get first element

```
>>> Stream(dict(a=3, b=4, c=5).items()).first()
('a', 3)
```

Returns

Return type element

first_opt()
Get first element as Some(element), or Nothing if not exists

Returns

Return type *Optional*[element]

flat_map(*to_iterable_func*)
Apply function to each element, then flatten the result.

```
>>> Stream([1, 2, 3]).flat_map(range).to_list()
[0, 0, 1, 0, 1, 2]
```

Returns

Return type *Stream*

flatten()
flatten each element

```
>>> Stream([(1, 2), (3, 4)]).flatten().to_list()
[1, 2, 3, 4]
```

Returns

Return type *Stream*

fold_left (*func*, *initial*)
Apply a function of two arguments cumulatively to the elements in Stream from left to right.

for_each (*func*)
Call function for each element

```
>>> s = Stream.range(3)
>>> s.for_each(print)
0
1
2
```

get (*index*, *default=None*)
Get item of the index. Return default value if not exists.

```
>>> s = Stream.range(5, 12)
>>> s.get(3)
8
>>> s.get(10) is None
True
>>> s.get(10, 0)
0
```

Returns

Return type element

get_opt (*index*)
Optionally get item of the index. Return Some(value) if exists. Otherwise return Nothing.

```
>>> s = Stream.range(5, 12)
>>> s.get_opt(3)
Some(8)
>>> s.get_opt(10)
Nothing
```

```
>>> s.get_opt(10).get_or(0)
0
>>> s.get_opt(3).map(lambda n: n * 2).get_or(0)
16
>>> s.get_opt(10).map(lambda n: n * 2).get_or(0)
0
```

Returns

Return type *Optional*[element]

group_by_as_map (*key_func=None*)
Group values in to a Map by the value of key function evaluation result.

Comparing to `group_by_as_stream`, there're some pros and cons.

Pros:

- Elements don't need to be sorted by the key function first. You can call `map_group_by` anytime and correct grouping result.

Cons:

- Key function has to be evaluated to a hashable value.
- Not Lazy-evaluating. Consume more memory while grouping. Yield a group as soon as possible.

```
>>> Stream.range(10).group_by_as_map(key_func=lambda n: n % 3)
Map({0: Array([0, 3, 6, 9]), 1: Array([1, 4, 7]), 2: Array([2, 5, 8])})
```

group_by_as_stream(*key=None*)

Create a new Stream using the builtin itertools.groupby, which sequentially groups elements as long as the key function evaluates to the same value.

Comparing to `group_by_as_map`, there're some pros and cons.

Cons:

- Elements should be sorted by the key function first, or elements with the same key may be broken into different groups.

Pros:

- Key function doesn't have to be evaluated to a hashable value. It can be any type which supports `__eq__`.
- Lazy-evaluating. Consume less memory while grouping. Yield a group as soon as possible.

interpose(*sep*)

Create a new Stream by interposing separater between elemens.

```
>>> Stream.range(5, 10).interpose(0).to_list()
[5, 0, 6, 0, 7, 0, 8, 0, 9]
```

classmethod iterate(*func*, *x*)

Create a Stream recursively applying a function to last return value.

```
>>> def multiply2(x): return x * 2
>>> Stream.iterate(multiply2, 3).take(4).to_list()
[3, 6, 12, 24]
```

last()

Get last element

Returns

Return type element

last_opt()

Get last element as Some(element), or Nothing if not exists

Returns

Return type *Optional*[element]

len()

Get the length of the Stream

Returns

Return type int

make_string(*elem_format='{elem!r}'*, *start='['*, *elem_sep=', '*, *end=']'*)

Make string from elements

```
>>> Stream.range(5, 8).make_string()
'[5, 6, 7]'
>>> print(Stream.range(5, 8).make_string(elem_sep='\n', start='', end='',␣
↪elem_format='{index}: {elem}'))
0: 5
1: 6
2: 7
```

map (*func*)
Create a new Stream by applying function to each element

```
>>> Stream.range(5, 8).map(lambda x: x * 2).to_list()
[10, 12, 14]
```

Returns

Return type *Stream*

mean ()
Get the average of elements.

```
>>> Array.range(10).mean()
4.5
```

nlargest (*n*, *key=None*)
Get the n largest elements.

```
>>> Stream([1, 5, 2, 3, 6]).nlargest(2).to_list()
[6, 5]
```

nsmallest (*n*, *key=None*)
Get the n smallest elements.

```
>>> Stream([1, 5, 2, 3, 6]).nsmallest(2).to_list()
[1, 2]
```

pluck (*key*)
Create a new Stream of values by evaluating `elem[key]` for each element.

```
>>> s = Stream([dict(x=3, y=4), dict(x=4, y=5), dict(x=8, y=9)])
>>> s.pluck('x').to_list()
[3, 4, 8]
```

Returns

Return type Stream[`element[key]`]

pluck_attr (*attr*)
Create a new Stream of Optional values by evaluating `elem.attr` of each element. Get `Some(value)` if attr exists for that element, otherwise get Nothing singleton.

```
>>> from carriage import Row
>>> s = Stream([Row(x=3, y=4), Row(x=4, y=5), Row(x=8, y=9)])
>>> s.pluck_attr('x').to_list()
[3, 4, 8]
```

Returns

Return type Stream[type of `element.attr`]

pluck_opt(*key*)
Create a new Stream of Optional values by evaluating `elem[key]` for each element. Get `Some(value)` if the key exists for that element, otherwise get Nothing singleton.

```
>>> s = Stream([dict(x=3, y=4), dict(y=5), dict(x=8, y=9)])
>>> s.pluck_opt('x').to_list()
[Some(3), Nothing, Some(8)]
>>> s.pluck_opt('x').map(lambda n_opt: n_opt.get_or(1)).to_list()
[3, 1, 8]
```

Returns

Return type Stream[Optional(type of `element[key]`)]

classmethod range(*start*, *end=None*, *step=1*)
Create a Stream from range.

```
>>> Stream.range(2, 10, 2).to_list()
[2, 4, 6, 8]
>>> Stream.range(3).to_list()
[0, 1, 2]
```

classmethod read_txt(*path*)
Create from a text file. Treat lines as elements and remove newline character.

```
>>> Stream.read_txt(path)
```

Parameters **path** (*str or path or file object*) – path to the input file

reduce(*func*)
Apply a function of two arguments cumulatively to the elements in Stream from left to right.

classmethod repeat(*elems*, *times=None*)
Create a Stream repeating elems

```
>>> Stream.repeat(1, 3).to_list()
[1, 1, 1]
>>> Stream.repeat([1, 2, 3], 2).to_list()
[[1, 2, 3], [1, 2, 3]]
```

classmethod repeatedly(*func*, *times=None*)
Create a Stream repeatedly calling a zero parameter function

```
>>> def counter():
...     counter.num += 1
...     return counter.num
>>> counter.num = -1
>>> Stream.repeatedly(counter, 5).to_list()
[0, 1, 2, 3, 4]
```

reversed()
Create a new reversed Stream.

```
>>> Stream(['a', 'b', 'c']).reversed().to_list()
['c', 'b', 'a']
```

second()
Get second element

```
>>> Stream(dict(a=3, b=4, c=5).items()).second()
('b', 4)
```

Returns

Return type element

second_opt()
Get second element as Some(element), or Nothing if not exists

Returns

Return type *Optional*[element]

show_pipeline(*n=2*)
Show pipeline and some examples for debugging

```
>>> def mul_2(x):
...     return x*2
>>> (Stream
...  .range(10)
...  .map(mul_2)
...  .nlargest(3)
...  .show_pipeline(2))
range(0, 10)
    [0] 0
    [1] 1
 -> map(<function mul_2 at 0x10a1dbd08>)
    [0] 0
    [1] 2
 -> nlargest(3)
    [0] 2
    [1] 0
```

slice(*start*, *stop*, *step=None*)
Create a Stream from the slice of items.

```
>>> Stream(list(range(10))).slice(5, 8).to_list()
[5, 6, 7]
```

Returns

Return type *Stream*[element]

sliding_window(*n*, *step=1*)
Create a new Stream instance that all elements are sliding windows of source elements.

```
>>> (Stream('they have the same meaning'.split())
...  .sliding_window(3)
...  .to_list())
[('they', 'have', 'the'), ('have', 'the', 'same'), ('the', 'same', 'meaning')]
```

```
>>> (Stream('they have the same meaning'.split())
...  .sliding_window(3, step=2)
...  .to_list())
[('they', 'have', 'the'), ('the', 'same', 'meaning')]
```

sorted(*key=None*, *reverse=False*)
Create a new sorted Stream.

split_after(*pred*)
Create a new Stream of Arrays by splitting after each element passing predicate.

```
>>> Stream.range(10).split_after(lambda n: n % 3 == 2).to_list()
[Array([0, 1, 2]), Array([3, 4, 5]), Array([6, 7, 8]), Array([9])]
```

split_before(*pred*)
Create a new Stream of Arrays by splitting before each element passing predicate.

```
>>> Stream.range(10).split_before(lambda n: n % 3 == 2).to_list()
[Array([0, 1]), Array([2, 3, 4]), Array([5, 6, 7]), Array([8, 9])]
```

star_for_each(*func*)
Call function for each element as agument tuple

```
>>> s = Stream(['a', 'b', 'c']).zip_index(1)
>>> s.star_for_each(lambda c, i: print(f'{i}:{c}'))
1:a
2:b
3:c
```

starmap(*func*)
Create a new Stream by evaluating function using argument tulpe from each element. i.e. `func(*elem)`. It's convenient that if all elements in Stream are iterable and you want to treat each element in elemnts as separate argument while calling the function.

```
>>> Stream([(1, 2), (3, 4)]).starmap(lambda a, b: a+b).to_list()
[3, 7]
>>> Stream([(1, 2), (3, 4)]).map(lambda a_b: a_b[0]+a_b[1]).to_list()
[3, 7]
```

sum()
Get sum of elements

tail()
Create a new Stream with first element dropped

```
>>> Stream(dict(a=3, b=4, c=5).items()).tail().to_list()
[('b', 4), ('c', 5)]
```

take(*n*)
Create a new Stream contains only first n element

```
>>> Stream(dict(a=3, b=4, c=5).items()).take(2).to_list()
[('a', 3), ('b', 4)]
```

take_while(*pred*)
Create a new Stream with successive elements as long as predicate evaluates to true.

```
>>> Stream.range(10).take_while(lambda n: n % 5 < 3).to_list()
[0, 1, 2]
```

takewhile (*pred*)
Create a new Stream with successive elements as long as predicate evaluates to true.

```
>>> Stream.range(10).take_while(lambda n: n % 5 < 3).to_list()
[0, 1, 2]
```

tap (*tag=''*, *n=5*, *msg_format='{tag}:{index}: {elem}'*)
A debugging tool. This method create a new Stream with the same elements. While evaluating Stream, it print first n elements.

```
>>> (Stream.range(3).tap('orig')
...  .map(lambda x: x * 2).tap_with(lambda i, e: f'{i} -> {e}')
...  .accumulate(lambda a, b: a + b).tap('acc')
...  .tap(msg_format='end\n')
...  .to_list())
orig:0: 0
0 -> 0
acc:0: 0
end

orig:1: 1
1 -> 2
acc:1: 2
end

orig:2: 2
2 -> 4
acc:2: 6
end

[0, 2, 6]
```

tap_with (*func*, *n=5*)
A debugging tool. This method create a new Stream with the same elements. While evaluating Stream, it call the function using index and element then prints the return value for first n elements.

```
>>> (Stream.range(3).tap('orig')
...  .map(lambda x: x * 2).tap('x2')
...  .accumulate(lambda a, b: a + b).tap('acc')
...  .to_list())
orig:0: 0
x2:0: 0
acc:0: 0
orig:1: 1
x2:1: 2
acc:1: 2
orig:2: 2
x2:2: 4
acc:2: 6
[0, 2, 6]
```

Parameters

- **func** (`func(index, elem) -> Any`) – Function for building the printing object.

- **n** (*int*) – First n element will be print.

tee(*n=2*)
Copy the Stream into multiple Stream with the same elements.

```
>>> itr = iter(range(3, 6))
>>> s1 = Stream(itr).map(lambda x: x * 2)
>>> s2, s3 = s1.tee(2)
>>> s2.map(lambda x: x * 2).to_list()
[12, 16, 20]
>>> s3.map(lambda x: x * 3).to_list()
[18, 24, 30]
```

to_array()
Convert to a Map

```
>>> Stream.range(5, 8, 2).zip_index().to_array()
Array([Row(value=5, index=0), Row(value=7, index=1)])
```

Returns

Return type *Array*

to_dict()
Convert to a dict

```
>>> Stream.range(5, 10, 2).zip_index().to_dict()
{5: 0, 7: 1, 9: 2}
```

Returns

Return type dict

to_list()
Convert to a list.

```
>>> Stream.range(5, 10, 2).to_list()
[5, 7, 9]
```

Returns

Return type list

to_map()
Convert to a Map

```
>>> Stream.range(5, 10, 2).zip_index().to_map()
Map({5: 0, 7: 1, 9: 2})
```

Returns

Return type *Map*

to_series()
Convert to a pandas Series

```
>>> Stream.range(5, 10, 2).to_series()
0    5
1    7
2    9
dtype: int64
```

Returns

Return type pandas.Series

to_set()
Convert to a set

```
>>> Stream.cycle([1, 2, 3]).take(5).to_set()
{1, 2, 3}
```

Returns

Return type set

to_streamtable()
Convert to StreamTable

All elements should be in Row type

Returns

Return type *StreamTable*

tuple_as_row(*fields*)
Create a new Stream with elements as Row objects

```
>>> Stream([(1, 2), (3, 4)]).tuple_as_row(['x', 'y']).to_list()
[Row(x=1, y=2), Row(x=3, y=4)]
```

unique(*key_func=None*)
Create a new Stream of unique elements

```
>>> Stream.range(10).unique(lambda x: x // 3).to_list()
[0, 3, 6, 9]
```

value_counts()
Get a Counter instance of elements counts

Returns

Return type *Map*[E, int]

without(**elems*)
Create a new Stream without specified elements.

```
>>> Stream.range(10).without(3, 6, 9).to_list()
[0, 1, 2, 4, 5, 7, 8]
```

Returns

Return type *Stream*[element]

write_txt (*path*, *sep='\n'*)
Write into a text file.

All elements will be applied `str()` before write to the file.

```
>>> Stream.range(10).write_txt('nums.txt')
```

path [str or path or file object] path to the input file

sep [str] element separator. defaults to '

'

zip (**iterables*)
Create a new Stream by zipping elements with other iterables.

```
>>> Stream.range(5, 8).zip([1,2,3]).to_list()
[Row(f0=5, f1=1), Row(f0=6, f1=2), Row(f0=7, f1=3)]
```

```
>>> Stream.range(5, 8).zip([1,2,3], [9, 10, 11]).to_list()
[Row(f0=5, f1=1, f2=9), Row(f0=6, f1=2, f2=10), Row(f0=7, f1=3, f2=11)]
```

```
>>> Stream.range(5, 8).zip([1,2]).to_list()
[Row(f0=5, f1=1), Row(f0=6, f1=2)]
```

```
>>> import itertools as itt
>>> Stream.range(5, 8).zip(itt.count(10)).to_list()
[Row(f0=5, f1=10), Row(f0=6, f1=11), Row(f0=7, f1=12)]
```

zip_index (*start=0*)
Create a new Stream by zipping elements with index.

```
>>> Stream(['a', 'b', 'c']).zip_index().to_list()
[Row(value='a', index=0), Row(value='b', index=1), Row(value='c', index=2)]
```

```
>>> Stream(['a', 'b', 'c']).zip_index(1).to_list()
[Row(value='a', index=1), Row(value='b', index=2), Row(value='c', index=3)]
```

zip_longest (**iterables*, *fillvalue=None*)
Create a new Stream by zipping elements with other iterables as long as possible.

```
>>> Stream.range(5, 8).zip_longest([1,2]).to_list()
[Row(f0=5, f1=1), Row(f0=6, f1=2), Row(f0=7, f1=None)]
```

```
>>> Stream.range(5, 8).zip_longest([1,2], fillvalue=0).to_list()
[Row(f0=5, f1=1), Row(f0=6, f1=2), Row(f0=7, f1=0)]
```

zip_next (*fillvalue=None*)
Create a new Stream by zipping elements with next one.

```
>>> Stream.range(5, 8).zip_next().to_list()
[Row(curr=5, prev=6), Row(curr=6, prev=7), Row(curr=7, prev=None)]
```

```
>>> Stream.range(5, 8).zip_next(fillvalue=1).to_list()
[Row(curr=5, prev=6), Row(curr=6, prev=7), Row(curr=7, prev=1)]
```

zip_prev (*fillvalue=None*)
Create a new Stream by zipping elements with previous one.

```
>>> Stream.range(5, 8).zip_prev().to_list()
[Row(curr=5, prev=None), Row(curr=6, prev=5), Row(curr=7, prev=6)]
```

```
>>> Stream.range(5, 8).zip_prev(fillvalue=0).to_list()
[Row(curr=5, prev=0), Row(curr=6, prev=5), Row(curr=7, prev=6)]
```

2.3 StreamTable: Lazy-evaluating sequential rows

class carriage.**StreamTable** (*iterable*, *, *pipeline=None*)
StreamTable is similar to Stream but designed to work on Rows only.

classmethod count (*start*, *step=1*)
Create a inifinite consecutive StreamTable

```
>>> StreamTable.count(3, 5).take(3).show()
|   count |
|---------|
|       3 |
|       8 |
|      13 |
```

classmethod cycle (*iterable*)
Create a StreamTable cycling a iterable

```
>>> StreamTable.cycle([1,2]).take(5).show()
|   cycle |
|---------|
|       1 |
|       2 |
|       1 |
|       2 |
|       1 |
```

explode (*field*)
Expand each row into multiple rows for each element in the field

```
>>> stb = StreamTable([Row(name='a', nums=[1,3,4]), Row(name='b', nums=[2,␣
↪1])])
>>> stb.explode('nums').show()
| name   |   nums |
|--------+--------|
| a      |      1 |
| a      |      3 |
| a      |      4 |
| b      |      2 |
| b      |      1 |
```

classmethod from_dataframe (*df*, *with_index=False*)
Create from Pandas DataFrame

```
>>> import pandas as pd
>>> df = pd.DataFrame([(0, 1), (2, 3)], columns=['a', 'b'])
```

(continues on next page)

(continued from previous page)

```
>>> StreamTable.from_dataframe(df).show()
|   a |   b |
|-----+-----|
|   0 |   1 |
|   2 |   3 |
```

Parameters

- **df** (*pandas.DataFrame*) – source DataFrame
- **with_index** (*bool*) – include index value or not

Returns

Return type *StreamTable*

classmethod from_tuples(*tuples*, *fields=None*)

Create from iterable of tuple

```
>>> StreamTable.from_tuples([(1, 2), (3, 4)], fields=('x', 'y')).show()
|   x |   y |
|-----+-----|
|   1 |   2 |
|   3 |   4 |
```

Parameters

- **tuples** (*Iterable[tuple]*) – data
- **fields** (*Tuple[str]*) – field names

classmethod iterate(*func*, *x*)

Create a StreamTable recursively applying a function to last return value.

```
>>> def multiply2(x): return x * 2
>>> StreamTable.iterate(multiply2, 3).take(4).show()
|   iterate |
|-----------|
|         3 |
|         6 |
|        12 |
|        24 |
```

map_fields(***field_funcs*)

Add or replace fields by applying each row to function

```
>>> from carriage import Row, X
>>> st = StreamTable([Row(x=3, y=4), Row(x=-1, y=2)])
>>> st.map_fields(z=X.x + X.y).to_list()
[Row(x=3, y=4, z=7), Row(x=-1, y=2, z=1)]
```

Parameters ****field_funcs** (*Map[field_name, Function]*) – Each function will be evaluated with the current row as the only argument, and the return value will be the new value of the field.

Returns

Return type *StreamTable*

classmethod range(*start*, *end=None*, *step=1*)
Create a StreamTable from range

```
>>> StreamTable.range(1, 10, 3).show()
|   range |
|---------|
|       1 |
|       4 |
|       7 |
```

classmethod read_jsonl(*path*)
Create from a jsonlines file

```
>>> StreamTable.read_jsonl('person.jsonl')
|   name |   age |
|--------+-------|
|   john |    18 |
|   jane |    26 |
```

Parameters **path** (*str or path or file object*) – path to the input file

classmethod repeat(*elems*, *times=None*)
Create a StreamTable repeating elems

```
>>> StreamTable.repeat(1, 3).show()
|   repeat |
|----------|
|        1 |
|        1 |
|        1 |
```

classmethod repeatedly(*func*, *times=None*)
Create a StreamTable repeatedly calling a zero parameter function

```
>>> def counter():
...     counter.num += 1
...     return counter.num
>>> counter.num = -1
>>> StreamTable.repeatedly(counter, 5).show()
|   repeatedly |
|--------------|
|            0 |
|            1 |
|            2 |
|            3 |
|            4 |
```

select(**fields*, ***field_funcs*)
Keep only specified fields, and add/replace fields.

```
>>> from carriage import Row, X
>>> st = StreamTable([Row(x=3, y=4), Row(x=-1, y=2)])
>>> st.select('x', z=X.x + X.y, pi=3.14).to_list()
[Row(x=3, z=7, pi=3.14), Row(x=-1, z=1, pi=3.14)]
```

Parameters

- ***fields** (*List[str]*) – fields to keep
- ****field_funcs** (*Map[str, Function or scalar]*) – If value is a function, this function will be evaluated with the current row as the only argument. If value is not callable, use the value directly.

Returns

Return type *StreamTable*

show(*n=10*)
print rows

Parameters **n** (*int*) – number of rows to show

tabulate(*n=10*, *tablefmt='orgtbl'*)
return tabulate formatted string

Parameters

- **n** (*int*) – number of rows to show
- **tablefmt** (*str*) – output table format. all possible format strings are in *StreamTable.tabulate.tablefmts'*

to_dataframe()
Convert to Pandas DataFrame

Returns

Return type pandas.DataFrame

to_stream()
Convert to Stream

Returns

Return type *Stream*

where(**conds*, ***kwconds*)
Create a new Stream contains only Rows pass all conditions.

```
>>> from carriage import Row, X
>>> st = StreamTable([Row(x=3, y=4), Row(x=3, y=5), Row(x=4, y=5)])
>>> st.where(x=3).to_list()
[Row(x=3, y=4), Row(x=3, y=5)]
>>> st.where(X.y > 4).to_list()
[Row(x=3, y=5), Row(x=4, y=5)]
```

Returns

Return type *StreamTable*

write_jsonl(*path*)
Write into file in the format of jsonlines

```
>>> stb.write_jsonl('person.jsonl')
```

Parameters **path** (*str or path or file object*) – path to the input file

2.4 X, Xcall: Elegant Lambda Function Builder

2.4.1 getitem

- `X['key']` equals to `lambda obj: obj['key']`

```
>>> from carriage import X, Stream
>>> stm = Stream([{'first name': 'John', 'last name': 'Doe', 'height': 180},
...               {'first name': 'Richard', 'last name': 'Roe', 'height': 190}])
>>> stm.map(X['first name']).to_list()
['John', 'Richard']
```

2.4.2 getattr

- `X.attr` equals to `lambda obj: obj.attr`

```
>>> from carriage import X, Stream, Row
>>> stm = Stream([Row(first_name='John', last_name='Doe', height=180),
...               Row(first_name='Richard', last_name='Roe', height=190)])
>>> stm.map(X.last_name).to_list()
['Doe', 'Roe']
```

2.4.3 Comparison operators

All comparison operators are supported: ==, !=, >, <, >=, <=

- `X > 3` equals to `lambda _: _ > 3`
- `'something' != X` equals to `lambda _: 'something' != _`

```
>>> from carriage import X, Stream, Row
>>> stm = Stream([Row(first_name='John', last_name='Doe', height=180),
...               Row(first_name='Richard', last_name='Roe', height=190),
...               Row(first_name='Jane', last_name='Doe', height=170)])
>>> stm.filter(X.height >= 180).to_list()
[Row(first_name='John', last_name='Doe', height=180), Row(first_name='Richard', last_
↪name='Roe', height=190)]
```

2.4.4 Math operators

All math and reflected math operators are supported: `X + Y`, `X - Y`, `X * Y`, `X / Y`, `X // Y`, `X % Y`, `divmod(X, Y)`, `X**Y`, `pow(X, Y)`, `abs(X)`, `+X`, `-X`

- `X + 3` equals to `lambda num: num + 3`
- `5 // X` equals to `lambda num: 5 // num`
- `pow(2, X)` equals to `lambda num: pow(2, num)`
- `divmod(X, 3)` equals to `lambda num: divmod(num, 2)`

```
>>> from carriage import X, Stream, Row
>>> stm = Stream([Row(x=5, y=3),
...               Row(x=9, y=3),
...               Row(x=3, y=8)])
>>> stm.map(X.x - X.y).to_list()
[2, 6, -5]
```

2.4.5 Method/Function calling

- `X.startswith.call('https')` equals to `lambda url:  url.startswith('https')`

```
>>> stm = Stream(['Callum', 'Reuben', 'Taylor', 'Lucas', 'Charles', 'Kylan', 'Camren',
↪ 'Edison', 'Raul'])
>>> stm.filter(X.startswith.call('C')).to_list()
['Callum', 'Charles', 'Camren']
```

2.4.6 As function arguments

- `Xcall(isinstance)(X, int)` equals to `lambda obj:  isinstance(obj, int)`

```
>>> import math
>>> from carriage import X, Stream, Row, Xcall
>>> stm = Stream([Row(x=5, y=3),
...               Row(x=9, y=3),
...               Row(x=3, y=8)])
>>> stm.map(Xcall(math.sqrt)(X.x**2 + X.y**2)).to_list()
[5.830951894845301, 9.486832980505138, 8.54400374531753]
```

2.4.7 Multiple X

- `X.height + X.width` equals to `lambda obj:  obj.height + obj.width`

2.4.8 In collection checking

- `X.in_((1,2))` equals to `lambda elem:  elem in (1, 2)`
- `X.has(1)` equals to `lambda coll:  1 in coll`

2.5 `Map`: Ordered dictionary with magic powers

class carriage.**Map**

A mutable dictionary enhanced with a bulk of useful methods.

filter(*pred*)

Create a new Map with key/value pairs satisfying the predicate

```
>>> m = Map({1: 2, 2: 4, 3: 6})
>>> m2 = m.filter(lambda k, v: (v-k) % 3 == 0)
>>> m2
Map({3: 6})
```

Parameters **pred** ((k, v) -> bool) – predicate

Returns

Return type *Map*[key, value]

filter_by_key(*pred*)

Create a new Map with keys satisfying the predicate

```
>>> m = Map({1: 2, 2: 4, 3: 6})
>>> m2 = m.filter_by_key(lambda k: k % 3 == 0)
>>> m2
Map({3: 6})
```

Parameters **pred** ((k, v) -> bool) – predicate

Returns

Return type *Map*[key, value]

filter_by_value(*pred*)

Create a new Map with values satisfying the predicate

```
>>> m = Map({1: 2, 2: 4, 3: 6})
>>> m2 = m.filter_by_value(lambda v: v % 3 == 0)
>>> m2
Map({3: 6})
```

Parameters **pred** ((k, v) -> bool) – predicate

Returns

Return type *Map*[key, value]

filter_false(*pred*)

Create a new Map with key/value pairs not satisfying the predicate

```
>>> m = Map({1: 2, 2: 4, 3: 6})
>>> m2 = m.filter_false(lambda k, v: (v-k) % 3 == 0)
>>> m2
Map({1: 2, 2: 4})
```

Parameters **pred** ((k, v) -> bool) – predicate

Returns

Return type *Map*[key, value]

first()

Get the first item in Row(key, value) type

```
>>> m = Map(a=4, b=5, c=6, d=7)
>>> m.first()
Row(key='a', value=4)
>>> m.first().key
'a'
>>> m.first().value
4
```

(continues on next page)

(continued from previous page)

```
>>> m = Map()
>>> m.first()
Traceback (most recent call last):
...
IndexError: index out of range.
```

Returns

Return type *Row*[key, value]

first_opt()

Optionally get the first item. Return Some(Row(key, value)) if first item exists, otherwise return Nothing

```
>>> m = Map(a=4, b=5, c=6, d=7)
>>> m.first_opt().map(lambda kv: kv.transform(value=lambda v: v * 2))
Some(Row(key='a', value=8))
>>> m.first_opt().map(lambda kv: kv.value)
Some(4)
>>> m = Map()
>>> m.first_opt()
Nothing
```

Returns

Return type *Optional*[*Row*[key, value]]

flip()

Create a new Map which key/value pairs are fliped

```
>>> m = Map(a=4, b=5, c=6)
>>> m.flip()
Map({4: 'a', 5: 'b', 6: 'c'})
```

for_each(*func*)

Call func for each key/value pair

```
>>> m = Map(a=[], b=[], c=[])
>>> m.for_each(lambda k, v: v.append(k))
>>> m
Map({'a': ['a'], 'b': ['b'], 'c': ['c']})
```

for_each_key(*func*)

Call func for each key

```
>>> m = Map(a=[], b=[], c=[])
>>> keys = []
>>> m.for_each_key(lambda k: keys.append(k))
>>> keys
['a', 'b', 'c']
```

for_each_value(*func*)

Call func for each value

```
>>> m = Map(a=[], b=[], c=[])
>>> m.for_each_value(lambda v: v.append(3))
```

(continues on next page)

(continued from previous page)

```
>>> m
Map({'a': [3], 'b': [3], 'c': [3]})
```

get_opt(*key*)

Get the value of specified key as Optional type. Return Some(value) if key exists, otherwise return Nothing.

```
>>> m = Map(a=3, b=4)
>>> m.get_opt('a')
Some(3)
>>> m.get_opt('c')
Nothing
>>> m.get_opt('a').map(lambda v: v * 2)
Some(6)
>>> m.get_opt('c').map(lambda v: v * 2)
Nothing
```

Returns

Return type *Optional*[value]

group_by(*key_func*)

Group key/value pairs into nested Maps.

```
>>> Map(a=3, b=4, c=5).group_by(lambda k, v: v % 2)
Map({1: Map({'a': 3, 'c': 5}), 0: Map({'b': 4})})
```

Parameters **key_func** ((key, value) -> group_key) – predicate

Returns

Return type *Map*[key_func(key), *Map*[key, value]]

items() → a set-like object providing a view on D's items

iter_joined(**others*, *fillvalue=None*, *agg=None*)

Create a `Row(key, Row(v0, v1, ...))` iterator with keys from all Maps and value joined.

```
>>> m = Map(a=1, b=2)
>>> l = list(m.iter_joined(
...           Map(a=3, b=4, c=5),
...           Map(a=6, c=7),
...           fillvalue=0))
>>> l[0]
Row(key='a', values=Row(f0=1, f1=3, f2=6))
>>> l[1]
Row(key='b', values=Row(f0=2, f1=4, f2=0))
>>> l[2]
Row(key='c', values=Row(f0=0, f1=5, f2=7))
```

join(**others*, *fillvalue=None*, *agg=None*)

Create a new Map instance with keys merged and values joined.

```
>>> m1 = Map(a=1, b=2)
>>> m2 = m1.join(dict(a=3, b=4, c=5))
>>> m2 is m1
False
```

(continues on next page)

(continued from previous page)

```
>>> m2
Map({'a': Row(f0=1, f1=3), 'b': Row(f0=2, f1=4), 'c': Row(f0=None, f1=5)})
```

```
>>> m1 = Map(a=1, b=2)
>>> m2 = m1.join(dict(a=3, b=4, c=5), agg=sum, fillvalue=0)
>>> m2
Map({'a': 4, 'b': 6, 'c': 5})
```

keep (**keys*)
Delete keys not specified and return self

```
>>> m = Map(a=3, b=4, c=5)
>>> m.keep('a', 'c')
Map({'a': 3, 'c': 5})
>>> m
Map({'a': 3, 'c': 5})
```

Returns

Return type self

keys () → a set-like object providing a view on D's keys

len ()
Get the length of this Map

```
>>> m = Map(a=4, b=5, c=6, d=7)
>>> m.len()
4
```

Returns

Return type int

make_string (*key_value_format='{key!r}: {value!r}'*, *start='{'*, *item_sep=', '*, *end='}'*)
Construct a string from key/values.

```
>>> m = Map(a=3, b=4, c=5)
>>> m.make_string()
"{'a': 3, 'b': 4, 'c': 5}"
>>> m.make_string(start='(', key_value_format='{key}={value!r}',
...               item_sep=', ', end=')')
'(a=3, b=4, c=5)'
```

Parameters

- **key_value_format** (*str*) – string template using builtin `str.format()` for formatting key/value pairs. Default to `'{key!r}: {value!r}'`. Available named placeholders: `{key}`, `{value}`
- **start** (*str*) – Default to `'{'`.
- **item_sep** (*str*) – Default to `', '`
- **end** (*str*) – Default to `}`

Returns

Return type str

map (*func*)
Create a new Map instance that each key, value pair is derived by applying function to original key, value.

```
>>> Map(a=3, b=4).map(lambda k, v: (v, k))
Map({3: 'a', 4: 'b'})
```

Parameters **func** (`pred(key, value) -> (key, value)`) – function for computing new key/value pair

map_keys (*func*)
Create a new Map instance that all values remains the same, while each corresponding key is updated by applying function to original key, value.

```
>>> Map(a=3, b=4).map_keys(lambda k, v: k + '_1')
Map({'a_1': 3, 'b_1': 4})
```

Parameters **func** (`pred(key, value) -> key`) – function for computing new keys

map_values (*func*)
Create a new Map instance that all keys remains the same, while each corresponding value is updated by applying function to original key, value.

```
>>> Map(a=3, b=4).map_values(lambda k, v: v * 2)
Map({'a': 6, 'b': 8})
```

Parameters **func** (`pred(key, value) -> value`) – function for computing new values

nlargest_value_items (*n=None*)
Get top n largest values

```
>>> m = Map(a=6, b=2, c=10, d=9)
>>> m.nlargest_value_items(n=2)
Array([Row(key='c', value=10), Row(key='d', value=9)])
```

Returns

Return type *Array*[*Row*[key, value]]

nsmallest_value_items (*n=None*)
Get top n smallest values

```
>>> m = Map(a=6, b=2, c=10, d=9)
>>> m.nsmallest_value_items(n=2)
Array([Row(key='b', value=2), Row(key='a', value=6)])
```

Returns

Return type *Array*[*Row*[key, value]]

nth (*index*)
Get the nth item in `Row(key, value)` type.

```
>>> m = Map(a=4, b=5, c=6, d=7)
>>> m.nth(2)
Row(key='c', value=6)
>>> m = Map(a=4, b=5)
>>> m.nth(2)
Traceback (most recent call last):
...
IndexError: index out of range.
```

Returns

Return type *Row*[key, value]

nth_opt(*index*)

Optionally get the nth item. Return `Some(Row(key, value))` if first item exists, otherwise return Nothing.

```
>>> m = Map(a=4, b=5, c=6, d=7)
>>> m.first_opt().map(lambda kv: kv.transform(value=lambda v: v * 2))
Some(Row(key='a', value=8))
>>> m = Map()
>>> m.first_opt()
Nothing
```

Returns

Return type *Optional*[*Row*[key, value]]

project(**keys*)

Create a new Map instance contains only specified keys.

```
>>> m = Map(a=3, b=4, c=5)
>>> m.project('a', 'c')
Map({'a': 3, 'c': 5})
>>> m
Map({'a': 3, 'b': 4, 'c': 5})
```

Returns

Return type *Map*[key, value]

remove(**keys*)

Delete keys and return self

```
>>> m = Map(a=3, b=4, c=5)
>>> m.remove('a', 'c')
Map({'b': 4})
>>> m
Map({'b': 4})
```

Returns

Return type self

retain(*pred*)

Delete key/value pairs not satisfying the predicate and return self

```
>>> m = Map(a=3, b=4, c=5)
>>> m.retain(lambda k, v: k == 'b' or v == 5)
Map({'b': 4, 'c': 5})
>>> m
Map({'b': 4, 'c': 5})
```

Parameters **pred** ((k, v) -> bool) –

Returns

Return type self

retain_by_key(*pred*)

Delete key/value pairs not satisfying the predicate and return self

```
>>> m = Map(a=3, b=4, c=5)
>>> m.retain_by_key(lambda k: k == 'b')
Map({'b': 4})
>>> m
Map({'b': 4})
```

Parameters **pred** ((k) -> bool) –

Returns

Return type self

retain_by_value(*pred*)

Delete key/value pairs not satisfying the predicate and return self

```
>>> m = Map(a=3, b=4, c=5)
>>> m.retain_by_value(lambda v: v == 4)
Map({'b': 4})
>>> m
Map({'b': 4})
```

Parameters **pred** ((k) -> bool) –

Returns

Return type self

retain_false(*pred*)

Delete key/value pairs satisfying the predicate and return self

```
>>> m = Map(a=3, b=4, c=5)
>>> m.retain_false(lambda k, v: k == 'b' or v == 5)
Map({'a': 3})
>>> m
Map({'a': 3})
```

Parameters **pred** ((k, v) -> bool) –

Returns

Return type self

revamp_values (*func*)
Update values of current Map and return self. Each value is derived by computing the function using both key and value.

```
>>> m = Map(a=3, b=4)
>>> m.revamp_values(lambda k, v: v * 2)
Map({'a': 6, 'b': 8})
>>> m
Map({'a': 6, 'b': 8})
```

Parameters **func** (`pred(key, value) -> value`) – function for computing new values

Returns

Return type self

take (*n*)
create a Stream instance of first `n` `Row(key, value)` elements.

```
>>> m = Map(a=4, b=5, c=6, d=7)
>>> m.take(2).to_list()
[Row(key='a', value=4), Row(key='b', value=5)]
```

Returns

Return type *Stream*[*Row*[key, value]]

to_array ()
Convert to an Array instance of `Row(key, value)` iterable.

```
>>> m = Map(a=4, b=5, c=6, d=7)
>>> m.to_array().take(2)
Array([Row(key='a', value=4), Row(key='b', value=5)])
```

Returns

Return type *Array*[*Row*[key, value]]

to_dict ()
Convert to dict

to_list ()
Convert to an list instance of `Row(key, value)` iterable.

```
>>> m = Map(a=4, b=5)
>>> m.to_list()
[Row(key='a', value=4), Row(key='b', value=5)]
```

Returns

Return type *Array*[*Row*[key, value]]

to_stream (*key_field='key'*, *value_field='value'*)
Convert to a Stream instance of `Row(key, value)` iterable.

```
>>> m = Map(a=4, b=5, c=6, d=7)
>>> m.to_stream().take(2).to_list()
[Row(key='a', value=4), Row(key='b', value=5)]
```

Returns

Return type *Stream*[*Row*[key, value]]

update(**args*, ***kwds*)
Update Map from dict/iterable and `return self`

```
>>> m = Map(a=3, b=4)
>>> m2 = m.update(a=5, c=3).update({'d': 2})
>>> m is m2
True
>>> m
Map({'a': 5, 'b': 4, 'c': 3, 'd': 2})
```

updated(**args*, ***kwds*)
Create a new Map instance that is updated from dict/iterable. This method is the same as `m.copy().update(...)`

```
>>> m = Map(a=3, b=4)
>>> m2 = m.updated(a=5, c=3).update({'d': 2})
>>> m2
Map({'a': 5, 'b': 4, 'c': 3, 'd': 2})
>>> m
Map({'a': 3, 'b': 4})
```

values() → an object providing a view on D's values

without(**keys*)
Create a new Map instance with those keys

```
>>> m = Map(a=3, b=4, c=6)
>>> m.without('a', 'c')
Map({'b': 4})
>>> m
Map({'a': 3, 'b': 4, 'c': 6})
```

Returns

Return type *Map*[key, value]

2.6 `Array`: All you want for a List type is here

class carriage.**Array**(*items=None*)

accumulate(*func=None*)
Create a new Array of calling `itertools.accumulate`

append(*item*)
Append element to the Array

appended(*item*)
Create a new Array that extends source Array with another element.

butlast()
Create a new Array that last element dropped

distincted()
Create a new Array with non-repeating elements. And elements are with the same order of first occurence in the source Array.

drop(*n*)
Create a new Array of first n element dropped

drop_right(*n*)
Create a new Array that last n elements dropped

drop_while(*pred*)
Create a new Array without elements as long as predicate evaluates to true.

dropright(*n*)
Create a new Array that last n elements dropped

dropwhile(*pred*)
Create a new Array without elements as long as predicate evaluates to true.

extend(*iterable*)
Extend the Array from iterable

extended(*iterable*)
Create a new Array that extends source Array with another iterable

filter(*pred*)
Create a new Array contains only elements passing predicate

filter_false(*pred*)
Create a new Array contains only elements not passing predicate

filterfalse(*pred*)
Create a new Array contains only elements not passing predicate

find(*pred*)
Get first element satifying predicate

find_opt(*pred*)
Optionally get first element satifying predicate. Return Some(element) if exist Otherwise return Nothing

first()
Get first element

first_opt()
Get first element as Some(element), or Nothing if not exists

flat_map(*to_iterable_action*)
Apply function to each element, then flatten the result.

```
>>> Array([1, 2, 3]).flat_map(range)
Array([0, 0, 1, 0, 1, 2])
```

Returns

Return type *Array*

flatten()
flatten each element

```
>>> Array([(1, 2), (3, 4)]).flatten()
Array([1, 2, 3, 4])
```

Returns

Return type *Array*

get(*index*, *default=None*)
Get item of the index. Return default value if not exists.

get_opt(*index*)
Optionally get item of the index. Return Some(value) if exists. Otherwise return Nothing.

group_by(*key=None*)
Create a new Array using the builtin itertools.groupby, which sequentially groups elements as long as the key function evaluates to the same value.

Comparing to `group_by_as_map`, there're some pros and cons.

Cons:

- Elements should be sorted by the key function first, or elements with the same key may be broken into different groups.

Pros:

- Key function doesn't have to be evaluated to a hashable value. It can be any type which supports `__eq__`.

group_by_as_map(*key=None*)
Group values in to a Map by the value of key function evaluation result.

Comparing to `group_by`, there're some pros and cons.

Pros:

- Elements don't need to be sorted by the key function first. You can call `map_group_by` anytime and correct grouping result.

Cons:

- Key function has to be evaluated to a hashable value.

interpose(*sep*)
Create a new Array by interposing separater between elemens.

last()
Get last element

last_opt()
Get last element as Some(element), or Nothing if not exists

len()
Get the length

make_string(*elem_format='{elem!r}'*, *start='['*, *elem_sep=', '*, *end=']'*)
Make string from elements

```
>>> Array.range(5, 8).make_string()
'[5, 6, 7]'
>>> print(Array.range(5, 8).make_string(elem_sep='\n', start='', end='', elem_
↪format='{index}: {elem}'))
0: 5
1: 6
2: 7
```

map(*action*)
Create a new Array by applying function to each element

```
>>> Array.range(5, 8).map(lambda x: x * 2)
Array([10, 12, 14])
```

Returns

Return type *Array*

mean()
Get the average of elements.

pluck(*key*)
Create a new Array of values by evaluating `elem[key]` for each element.

pluck_attr(*attr*)
Create a new Array of Optional values by evaluating `elem.attr` of each element. Get `Some(value)` if attr exists for that element, otherwise get Nothing singleton.

pluck_opt(*key*)
Create a new Array of Optional values by evaluating `elem[key]` for each element. Get `Some(value)` if the key exists for that element, otherwise get Nothing singleton.

classmethod range(*start*, *end=None*, *step=1*)
Create a Array from range.

```
>>> Array.range(2, 10, 2).to_list()
[2, 4, 6, 8]
>>> Array.range(3).to_list()
[0, 1, 2]
```

reverse()
In place reverse this Array.

reversed()
Create a new reversed Array.

second()
Get second element

second_opt()
Get second element as Some(element), or Nothing if not exists

slice(*start*, *stop*, *step=None*)
Create a Array from the slice of items.

Returns

Return type *Array*

sliding_window(*n*)
Create a new Array instance that all elements are sliding windows of source elements.

sort(*key=None*, *reverse=False*)
In place sort this Array.

sorted(*key=None*, *reverse=False*)
Create a new sorted Array.

split_after(*pred*)
Create a new Array of Arrays by splitting after each element passing predicate.

split_before(*pred*)
Create a new Array of Arrays by splitting before each element passing predicate.

starmap(*func*)
Create a new Array by evaluating function using argument tulpe from each element. i.e. `func(*elem)`. It's convenient that if all elements in Array are iterable and you want to treat each element in elemnts as separate argument while calling the function.

```
>>> Array([(1, 2), (3, 4)]).starmap(lambda a, b: a+b)
Array([3, 7])
```

The map way. Not easy to read and write

```
>>> Array([(1, 2), (3, 4)]).map(lambda a_b: a_b[0]+a_b[1])
Array([3, 7])
```

sum()
Get sum of elements

tail()
Create a new Array first element dropped

take(*n*)
Create a new Array of only first n element

take_right(*n*)
Create a new Array with last n elements

take_while(*pred*)
Create a new Array with successive elements as long as predicate evaluates to true.

takeright(*n*)
Create a new Array with last n elements

takewhile(*pred*)
Create a new Array with successive elements as long as predicate evaluates to true.

tap(*tag=''*, *n=5*, *msg_format='{tag}:{index}: {elem}'*)
A debugging tool. This method create a new Array with the same elements. While creating it, it print first n elements.

```
>>> (Array.range(3).tap('orig')
...  .map(lambda x: x * 2).tap('x2')
...  .accumulate(lambda a, b: a + b)
...  .tap_with(func=lambda i, e: f'{i} -> {e}')
... )
orig:0: 0
orig:1: 1
orig:2: 2
```

(continues on next page)

(continued from previous page)

```
x2:0: 0
x2:1: 2
x2:2: 4
0 -> 0
1 -> 2
2 -> 6
Array([0, 2, 6])
```

tap_with (*func*, *n=5*)

A debugging tool. This method create a new Array with the same elements. While creating Array, it call the function using index and element then prints the return value for first n elements.

```
>>> (Array.range(3).tap('orig')
...  .map(lambda x: x * 2).tap('x2')
...  .accumulate(lambda a, b: a + b)
...  .tap_with(func=lambda i, e: f'{i} -> {e}')
... )
orig:0: 0
orig:1: 1
orig:2: 2
x2:0: 0
x2:1: 2
x2:2: 4
0 -> 0
1 -> 2
2 -> 6
Array([0, 2, 6])
```

Parameters

- **func** (`func(index, elem) -> Any`) – Function for building the printing object.
- **n** (*int*) – First n element will be print.

to_dict()

Convert to a dict

```
>>> Array.range(5, 10, 2).zip_index().to_dict()
{5: 0, 7: 1, 9: 2}
```

Returns

Return type dict

to_list (*copy=False*)

Convert to a list.

```
>>> Array.range(3).to_list()
[0, 1, 2]
```

to_map()

Convert to a Map

```
>>> Array.range(5, 10, 2).zip_index().to_map()
Map({5: 0, 7: 1, 9: 2})
```

Returns

Return type *Map*

to_series()
Convert to a pandas Series

```
>>> Array.range(5, 10, 2).to_series()
0    5
1    7
2    9
dtype: int64
```

Returns

Return type pandas.Series

to_set()
Convert to a set

```
>>> Array([3, 2, 3, 6, 2]).to_set()
{2, 3, 6}
```

Returns

Return type set

to_stream()
Convert to a Stream

```
>>> strm = Array.range(5, 8, 2).zip_index().to_stream()
>>> type(strm)
<class 'carriage.stream.Stream'>
>>> strm.to_array()
Array([Row(value=5, index=0), Row(value=7, index=1)])
```

Returns

Return type *Stream*

value_counts()
Get a Counter instance of elements counts

where(***conds*)
Create a new Array contains only mapping pass all conditions.

without(**items*)
Create a new Array without specified elements.

zip(**iterable*)
Create a new Array by zipping elements with other iterables.

zip_index(*start=0*)
Create a new Array by zipping elements with index.

zip_longest(**iterables*, *fillvalue=None*)
Create a new Array by zipping elements with other iterables as long as possible.

zip_next (*fillvalue=None*)
Create a new Array by zipping elements with next one.

zip_prev (*fillvalue=None*)
Create a new Array by zipping elements with previous one.

2.7 `Optional`: Object wrapper for handling errors

class carriage.**Optional**
An type for handling special value or exception.

Here is a contacts data constructed with multiple levels dictionary.

```
>>> contacts = {
...     'John Doe': {
...         'phone': '0911-222-333',
...         'address': {'city': 'hsinchu',
...                     'street': '185 Somewhere St.'}},
...     'Richard Roe': {
...         'phone': '0933-444-555',
...         'address': {'city': None,
...                     'street': None}},
...     'Mark Moe': {
...         'address': None},
...     'Larry Loe': None
... }
```

If we need a function to get the formatted city name of some contact, we will have a lot of nested *if* statement for handling None or other unexpected values.

```
>>> def get_city(name):
...     contact = contacts.get(name)
...     if contact is not None:
...         address = contact.get('address')
...         if address is not None:
...             city = address.get('city')
...             if city is not None:
...                 return f'City: {city}'
...
...     return 'No city available'
>>> get_city('John Doe')
'City: hsinchu'
>>> get_city('Richard Roe')
'No city available'
>>> get_city('Mark Moe')
'No city available'
>>> get_city('Larray Loe')
'No city available'
>>> get_city('Not Existing')
'No city available'
```

Optional is useful on handling unexpected return values or exceptions and makes the code shorter and more readable.

```
>>> def getitem_opt(obj, key):
...     """The same as Optional.from_getitem()"""
```

(continues on next page)

(continued from previous page)

```
...     try:
...         return Some(obj[key])
...     except (KeyError, TypeError):
...         return Nothing
...
>>> def get_city2(name):
...     return (getitem_opt(contacts, name)
...             .and_then(lambda contact: getitem_opt(contact, 'address'))
...             .and_then(lambda address: getitem_opt(address, 'city'))
...             .filter(lambda city: city is not None)
...             .map(lambda city: f'City: {city}')
...             .get_or('No city available')
...             )
...
>>> get_city2('John Doe')
'City: hsinchu'
>>> get_city2('Richard Roe')
'No city available'
>>> get_city2('Mark Moe')
'No city available'
>>> get_city2('Larray Loe')
'No city available'
>>> get_city('Not Existing')
'No city available'
```

Create Optional directly

```
>>> Some(3)
Some(3)
>>> Nothing
Nothing
```

Create Optional by calling a function that may throw exception

```
>>> def divide(a, b):
...     return a / b
>>> Optional.from_call(divide, 2, 4, errors=ZeroDivisionError)
Some(0.5)
>>> Optional.from_call(divide, 2, 0, errors=ZeroDivisionError)
Nothing
```

Create Optional from a value that may be None or other spectial value.

```
>>> adict = {'a': 1, 'b': 2, 'c': 3}
>>> Optional.from_value(adict.get('c'), nothing_value=None)
Some(3)
>>> Optional.from_value(adict.get('d'), nothing_value=None)
Nothing
```

and_then (*optional_func*)

Return `optional_func(value)` if it is Some `optional_func` should return Optional

`and_then` is useful for chaining functions that return Optional

filter (*pred*)

Return Nothing if Some doesn't satisfy the predicate

classmethod from_call(*func*, **args*, *errors=(<class 'Exception'>,)*, ***kwargs*)
Create an Optional by calling a function

return Nothing if exception is raised

classmethod from_getattr(*obj*, *attr_name*)
Create an Optional by calling `obj.attr_name`

return Nothing if AttributeError is raised

classmethod from_getitem(*obj*, *key*)
Create an Optional by calling `obj[key]`

return `Nothing` if `KeyError` or `TypeError` is raised

classmethod from_value(*value*, *nothing_value=None*)
Create an Optional from a value

return Nothing if `value` equals to `nothing_value`

get_or(*default*)
Get the value if it is Some or get *default* if it is Nothing

get_or_none()
Get the value if it is Some or get None if it is Nothing

is_nothing()
Check if it is Nothing

is_some()
Check if it is Some

map(*func*)
Return `Some(func(value))` if it is Some

some
Get the value if it is Some or raise AttributeError if it is not

carriage.**Nothing**

CHAPTER 3

To Do

- A simple lambda function generating type.
- Multi-core processing.
- I/O methods for reading and writing to files.

CHAPTER 4

Indices and tables

- genindex
- modindex
- search

Index