Pyrsistent Documentation

Release 0.X.Y (moving target)

Tobias Gustafsson

Oct 08, 2017

Contents

Contents:

CHAPTER 1

Pyrsistent

Pyrsistent is a number of persistent collections (by some referred to as functional data structures). Persistent in the sense that they are immutable.

All methods on a data structure that would normally mutate it instead return a new copy of the structure containing the requested updates. The original structure is left untouched.

This will simplify the reasoning about what a program does since no hidden side effects ever can take place to these data structures. You can rest assured that the object you hold a reference to will remain the same throughout its lifetime and need not worry that somewhere five stack levels below you in the darkest corner of your application someone has decided to remove that element that you expected to be there.

Pyrsistent is influenced by persistent data structures such as those found in the standard library of Clojure. The data structures are designed to share common elements through path copying. It aims at taking these concepts and make them as pythonic as possible so that they can be easily integrated into any python program without hassle.

If you want to go all in on persistent data structures and use literal syntax to define them in your code rather than function calls check out Pyrthon.

Examples

The collection types and key features currently implemented are:

- *PVector*, similar to a python list
- *PMap*, similar to dict
- *PSet*, similar to set
- *PRecord*, a PMap on steroids with fixed fields, optional type and invariant checking and much more
- *PClass*, a Python class fixed fields, optional type and invariant checking and much more
- *Checked collections*, PVector, PMap and PSet with optional type and invariance checks and more
- PBag, similar to collections.Counter

- PList, a classic singly linked list
- PDeque, similar to collections.deque
- Immutable object type (immutable) built on the named tuple
- *freeze* and *thaw* functions to convert between pythons standard collections and pyrsistent collections.
- Flexible *transformations* of arbitrarily complex structures built from PMaps and PVectors.

Below are examples of common usage patterns for some of the structures and features. More information and full documentation for all data structures is available in the documentation.

PVector

With full support for the Sequence protocol PVector is meant as a drop in replacement to the built in list from a readers point of view. Write operations of course differ since no in place mutation is done but naming should be in line with corresponding operations on the built in list.

Support for the Hashable protocol also means that it can be used as key in Mappings.

Appends are amortized O(1). Random access and insert is log32(n) where n is the size of the vector.

```
>>> from pyrsistent import v, pvector

# No mutation of vectors once created, instead they
# are "evolved" leaving the original untouched
>>> v1 = v(1, 2, 3)
>>> v2 = v1.append(4)
>>> v3 = v2.set(1, 5)
>>> v1
pvector([1, 2, 3])
>>> v2
pvector([1, 2, 3, 4])
>>> v3
pvector([1, 5, 3, 4])

# Random access and slicing
>>> v3[1]
5
>>> v3[1:3]
pvector([5, 3])

# Iteration
>>> list(x + 1 for x in v3)
[2, 6, 4, 5]
>>> pvector(2 * x for x in range(3))
pvector([0, 2, 4])
```

PMap

With full support for the Mapping protocol PMap is meant as a drop in replacement to the built in dict from a readers point of view. Support for the Hashable protocol also means that it can be used as key in other Mappings.

Random access and insert is log32(n) where n is the size of the map.

```
>>> from pyrsistent import m, pmap, v

```

```
# No mutation of maps once created, instead they are
# "evolved" leaving the original untouched
>>> m1 = m(a=1, b=2)
>>> m2 = m1.set('c', 3)
>>> m3 = m2.set('a', 5)
>>> m1
pmap({'a': 1, 'b': 2})
>>> m2
pmap({'a': 1, 'c': 3, 'b': 2})
>>> m3
pmap({'a': 5, 'c': 3, 'b': 2})
>>> m3['a']
5

# Evolution of nested persistent structures
>>> m4 = m(a=5, b=6, c=v(1, 2))
>>> m4.transform(('c', 1), 17)
pmap({'a': 5, 'c': pvector([1, 17]), 'b': 6})
>>> m5 = m(a=1, b=2)

# Evolve by merging with other mappings
>>> m5.update(m(a=2, c=3), {'a': 17, 'd': 35})
pmap({'a': 17, 'c': 3, 'b': 2, 'd': 35})
>>> pmap({'x': 1, 'y': 2}) + pmap({'y': 3, 'z': 4})
pmap({'y': 3, 'x': 1, 'z': 4})

# Dict-like methods to convert to list and iterate
>>> m3.items()
pvector([('a', 5), ('c', 3), ('b', 2)])
>>> list(m3)
['a', 'c', 'b']
```

PSet

With full support for the Set protocol PSet is meant as a drop in replacement to the built in set from a readers point of view. Support for the Hashable protocol also means that it can be used as key in Mappings.

Random access and insert is log32(n) where n is the size of the set.

```
>>> from pyrsistent import s

# No mutation of sets once created, you know the story...
>>> s1 = s(1, 2, 3, 2)
>>> s2 = s1.add(4)
>>> s3 = s1.remove(1)
>>> s1
pset([1, 2, 3])
>>> s2
pset([1, 2, 3, 4])
>>> s3
pset([2, 3])

# Full support for set operations
>>> s1 | s(3, 4, 5)
pset([1, 2, 3, 4, 5])
>>> s1 & s(3, 4, 5)
pset([3])
```

```
>>> s1 < s2
True
>>> s1 < s(3, 4, 5)
False
```

PRecord

A PRecord is a PMap with a fixed set of specified fields. Records are declared as python classes inheriting from PRecord. Because it is a PMap it has full support for all Mapping methods such as iteration and element access using subscript notation.

```
>>> from pyrsistent import PRecord, field
>>> class ARecord(PRecord):
...     x = field()
...
>>> r = ARecord(x=3)
>>> r
ARecord(x=3)
>>> r.x
3
>>> r.set(x=2)
ARecord(x=2)
>>> r.set(y=2)
Traceback (most recent call last):
AttributeError: 'y' is not among the specified fields for ARecord
```

Type information

It is possible to add type information to the record to enforce type checks. Multiple allowed types can be specified by providing an iterable of types.

```
>>> class BRecord(PRecord):
...     x = field(type=int)
...     y = field(type=(int, type(None)))
...
>>> BRecord(x=3, y=None)
BRecord(y=None, x=3)
>>> BRecord(x=3.0)
Traceback (most recent call last):
PTypeError: Invalid type for field BRecord.x, was float
```

Custom types (classes) that are iterable should be wrapped in a tuple to prevent their members being added to the set of valid types. Although Enums in particular are now supported without wrapping, see #83 for more information.

Mandatory fields

Fields are not mandatory by default but can be specified as such. If fields are missing an *InvariantException* will be thrown which contains information about the missing fields.

```
>>> from pyrsistent import InvariantException
>>> class CRecord(PRecord):
...     x = field(mandatory=True)
...
```

```
>>> r = CRecord(x=3)
>>> try:
...     r.discard('x')
... except InvariantException as e:
...     print(e.missing_fields)
...
('CRecord.x',)
```

Invariants

It is possible to add invariants that must hold when evolving the record. Invariants can be specified on both field and record level. If invariants fail an *InvariantException* will be thrown which contains information about the failing invariants. An invariant function should return a tuple consisting of a boolean that tells if the invariant holds or not and an object describing the invariant. This object can later be used to identify which invariant that failed.

The global invariant function is only executed if all field invariants hold.

Global invariants are inherited to subclasses.

```
>>> class RestrictedVector(PRecord):
...      __invariant__ = lambda r: (r.y >= r.x, 'x larger than y')
...      x = field(invariant=lambda x: (x > 0, 'x negative'))
...      y = field(invariant=lambda y: (y > 0, 'y negative'))
...
>>> r = RestrictedVector(y=3, x=2)
>>> try:
...     r.set(x=-1, y=-2)
... except InvariantException as e:
...     print(e.invariant_errors)
...
('y negative', 'x negative')
>>> try:
...     r.set(x=2, y=1)
... except InvariantException as e:
...     print(e.invariant_errors)
...
('x larger than y',)
```

Invariants may also contain multiple assertions. For those cases the invariant function should return a tuple of invariant tuples as described above. This structure is reflected in the invariant_errors attribute of the exception which will contain tuples with data from all failed invariants. Eg:

```
>>> class EvenX(PRecord):
...      x = field(invariant=lambda x: ((x > 0, 'x negative'), (x % 2 == 0, 'x odd')))
...
>>> try:
...     EvenX(x=-1)
... except InvariantException as e:
...     print(e.invariant_errors)
...
(('x negative', 'x odd'),)
```

Factories

It's possible to specify factory functions for fields. The factory function receives whatever is supplied as field value and the actual returned by the factory is assigned to the field given that any type and invariant checks hold. PRecords have a default factory specified as a static function on the class, create(). It takes a *Mapping* as argument and returns an instance of the specific record. If a record has fields of type PRecord the create() method of that record will be called to create the "sub record" if no factory has explicitly been specified to override this behaviour.

```
>>> class DRecord(PRecord):
...     x = field(factory=int)
...
>>> class ERecord(PRecord):
...     d = field(type=DRecord)
...
>>> ERecord.create({'d': {'x': '1'}})
ERecord(d=DRecord(x=1))
```

Collection fields

It is also possible to have fields with `pyrsistent` collections.

```
>>> from pyrsistent import pset_field, pmap_field, pvector_field
>>> class MultiRecord(PRecord):
...     set_of_ints = pset_field(int)
...     map_int_to_str = pmap_field(int, str)
...     vector_of_strs = pvector_field(str)
...
```

Serialization

PRecords support serialization back to dicts. Default serialization will take keys and values "as is" and output them into a dict. It is possible to specify custom serialization functions to take care of fields that require special treatment.

```
>>> from datetime import date
>>> class Person(PRecord):
...     name = field(type=unicode)
...     birth_date = field(type=date,
...                        serializer=lambda format, d: d.strftime(format['date']))
...
>>> john = Person(name=u'John', birth_date=date(1985, 10, 21))
>>> john.serialize({'date': '%Y-%m-%d'})
{'birth_date': '1985-10-21', 'name': u'John'}
```

PClass

A PClass is a python class with a fixed set of specified fields. PClasses are declared as python classes inheriting from PClass. It is defined the same way that PRecords are and behaves like a PRecord in all aspects except that it is not a PMap and hence not a collection but rather a plain Python object.

```
>>> from pyrsistent import PClass, field
>>> class AClass(PClass):
...     x = field()
...
```

```
>>> a = AClass(x=3)
>>> a
AClass(x=3)
>>> a.x
3
```

Checked collections

Checked collections currently come in three flavors: CheckedPVector, CheckedPMap and CheckedPSet.

```
>>> from pyrsistent import CheckedPVector, CheckedPMap, CheckedPSet, thaw
>>> class Positives(CheckedPSet):
...     __type__ = (long, int)
...     __invariant__ = lambda n: (n >= 0, 'Negative')
...
>>> class Lottery(PRecord):
...     name = field(type=str)
...     numbers = field(type=Positives, invariant=lambda p: (len(p) > 0, 'No numbers
↪'))
...
>>> class Lotteries(CheckedPVector):
...     __type__ = Lottery
...
>>> class LotteriesByDate(CheckedPMap):
...     __key_type__ = date
...     __value_type__ = Lotteries
...
>>> lotteries = LotteriesByDate.create({date(2015, 2, 15): [{'name': 'SuperLotto',
↪'numbers': {1, 2, 3}},
...                                                         {'name': 'MegaLotto',
↪'numbers': {4, 5, 6}}],
...                                     date(2015, 2, 16): [{'name': 'SuperLotto',
↪'numbers': {3, 2, 1}},
...                                                         {'name': 'MegaLotto',
↪'numbers': {6, 5, 4}}]})
>>> lotteries
LotteriesByDate({datetime.date(2015, 2, 15): Lotteries([Lottery(numbers=Positives([1,␣
↪2, 3]), name='SuperLotto'), Lottery(numbers=Positives([4, 5, 6]), name='MegaLotto
↪')]), datetime.date(2015, 2, 16): Lotteries([Lottery(numbers=Positives([1, 2, 3]),␣
↪name='SuperLotto'), Lottery(numbers=Positives([4, 5, 6]), name='MegaLotto')])})

# The checked versions support all operations that the corresponding
# unchecked types do
>>> lottery_0215 = lotteries[date(2015, 2, 15)]
>>> lottery_0215.transform([0, 'name'], 'SuperDuperLotto')
Lotteries([Lottery(numbers=Positives([1, 2, 3]), name='SuperDuperLotto'),␣
↪Lottery(numbers=Positives([4, 5, 6]), name='MegaLotto')])

# But also makes asserts that types and invariants hold
>>> lottery_0215.transform([0, 'name'], 999)
Traceback (most recent call last):
PTypeError: Invalid type for field Lottery.name, was int

>>> lottery_0215.transform([0, 'numbers'], set())
Traceback (most recent call last):
InvariantException: Field invariant failed
```

```
# They can be converted back to python built ins with either thaw()
# or serialize() (which provides possibilities to customize serialization)
>>> thaw(lottery_0215)
[{'numbers': set([1, 2, 3]), 'name': 'SuperLotto'}, {'numbers': set([4, 5, 6]), 'name
↪': 'MegaLotto'}]
>>> lottery_0215.serialize()
[{'numbers': set([1, 2, 3]), 'name': 'SuperLotto'}, {'numbers': set([4, 5, 6]), 'name
↪': 'MegaLotto'}]
```

Transformations

Transformations are inspired by the cool library instar for Clojure. They let you evolve PMaps and PVectors with arbitrarily deep/complex nesting using simple syntax and flexible matching syntax.

The first argument to transformation is the path that points out the value to transform. The second is the transformation to perform. If the transformation is callable it will be applied to the value(s) matching the path. The path may also contain callables. In that case they are treated as matchers. If the matcher returns True for a specific key it is considered for transformation.

```
# Basic examples
>>> from pyrsistent import inc, freeze, thaw, rex, ny, discard
>>> v1 = freeze([1, 2, 3, 4, 5])
>>> v1.transform([2], inc)
pvector([1, 2, 4, 4, 5])
>>> v1.transform([lambda ix: 0 < ix < 4], 8)
pvector([1, 8, 8, 8, 5])
>>> v1.transform([lambda ix, v: ix == 0 or v == 5], 0)
pvector([0, 2, 3, 4, 0])

# The (a)ny matcher can be used to match anything
>>> v1.transform([ny], 8)
pvector([8, 8, 8, 8, 8])

# Regular expressions can be used for matching
>>> scores = freeze({'John': 12, 'Joseph': 34, 'Sara': 23})
>>> scores.transform([rex('^Jo')], 0)
pmap({'Joseph': 0, 'Sara': 23, 'John': 0})

# Transformations can be done on arbitrarily deep structures
>>> news_paper = freeze({'articles': [{'author': 'Sara', 'content': 'A short article'}
↪,
...                                   {'author': 'Steve', 'content': 'A slightly␣
↪longer article'}],
...                         'weather': {'temperature': '11C', 'wind': '5m/s'}})
>>> short_news = news_paper.transform(['articles', ny, 'content'], lambda c: c[:25] +
↪'...' if len(c) > 25 else c)
>>> very_short_news = news_paper.transform(['articles', ny, 'content'], lambda c:␣
↪c[:15] + '...' if len(c) > 15 else c)
>>> very_short_news.articles[0].content
'A short article'
>>> very_short_news.articles[1].content
'A slightly long...'

# When nothing has been transformed the original data structure is kept
>>> short_news is news_paper
```

```
True
>>> very_short_news is news_paper
False
>>> very_short_news.articles[0] is news_paper.articles[0]
True

# There is a special transformation that can be used to discard elements. Also
# multiple transformations can be applied in one call
>>> thaw(news_paper.transform(['weather'], discard, ['articles', ny, 'content'],␣
↪discard))
{'articles': [{'author': 'Sara'}, {'author': 'Steve'}]}
```

Evolvers

PVector, PMap and PSet all have support for a concept dubbed *evolvers*. An evolver acts like a mutable view of the underlying persistent data structure with "transaction like" semantics. No updates of the original data structure is ever performed, it is still fully immutable.

The evolvers have a very limited API by design to discourage excessive, and inappropriate, usage as that would take us down the mutable road. In principle only basic mutation and element access functions are supported. Check out the documentation of each data structure for specific examples.

Examples of when you may want to use an evolver instead of working directly with the data structure include:

- Multiple updates are done to the same data structure and the intermediate results are of no interest. In this case using an evolver may be a more efficient and easier to work with.
- You need to pass a vector into a legacy function or a function that you have no control over which performs in place mutations. In this case pass an evolver instance instead and then create a new pvector from the evolver once the function returns.

```
>>> from pyrsistent import v

# In place mutation as when working with the built in counterpart
>>> v1 = v(1, 2, 3)
>>> e = v1.evolver()
>>> e[1] = 22
>>> e = e.append(4)
>>> e = e.extend([5, 6])
>>> e[5] += 1
>>> len(e)
6

# The evolver is considered *dirty* when it contains changes compared to the␣
↪underlying vector
>>> e.is_dirty()
True

# But the underlying pvector still remains untouched
>>> v1
pvector([1, 2, 3])

# Once satisfied with the updates you can produce a new pvector containing the␣
↪updates.
# The new pvector will share data with the original pvector in the same way that␣
↪would have
# been done if only using operations on the pvector.
```

```
>>> v2 = e.persistent()
>>> v2
pvector([1, 22, 3, 4, 5, 7])

# The evolver is now no longer considered *dirty* as it contains no differences␣
↪compared to the
# pvector just produced.
>>> e.is_dirty()
False

# You may continue to work with the same evolver without affecting the content of v2
>>> e[0] = 11

# Or create a new evolver from v2. The two evolvers can be updated independently but␣
↪will both
# share data with v2 where possible.
>>> e2 = v2.evolver()
>>> e2[0] = 1111
>>> e.persistent()
pvector([11, 22, 3, 4, 5, 7])
>>> e2.persistent()
pvector([1111, 22, 3, 4, 5, 7])
```

freeze and thaw

These functions are great when your cozy immutable world has to interact with the evil mutable world outside.

```
>>> from pyrsistent import freeze, thaw, v, m
>>> freeze([1, {'a': 3}])
pvector([1, pmap({'a': 3})])
>>> thaw(v(1, m(a=3)))
[1, {'a': 3}]
```

Compatibility

Pyrsistent is developed and tested on Python 2.6, 2.7, 3.4, 3.5 and PyPy (Python 2.7 compatible). It will most likely work on all other versions >= 3.4 but no guarantees are given. :)

Compatibility issues

There is currently one known compatibility issue when comparing built in sets and frozensets to PSets as discussed in 27. It affects python 2 versions < 2.7.8 and python 3 versions < 3.4.0 and is due to a bug described in http://bugs.python.org/issue8743.

Comparisons will fail or be incorrect when using the set/frozenset as left hand side of the comparison. As a workaround you need to either upgrade Python to a more recent version, avoid comparing sets/frozensets with PSets or always make sure to convert both sides of the comparison to the same type before performing the comparison.

Performance

Pyrsistent is developed with performance in mind. Still, while some operations are nearly on par with their built in, mutable, counterparts in terms of speed, other operations are slower. In the cases where attempts at optimizations have been done, speed has generally been valued over space.

Pyrsistent comes with two API compatible flavors of PVector (on which PMap and PSet are based), one pure Python implementation and one implemented as a C extension. The latter generally being 2 - 20 times faster than the former. The C extension will be used automatically when possible.

The pure python implementation is fully PyPy compatible. Running it under PyPy speeds operations up considerably if the structures are used heavily (if JITed), for some cases the performance is almost on par with the built in counterparts.

Installation

pip install pyrsistent

Documentation

Available at http://pyrsistent.readthedocs.org/

Brief presentation available at http://slides.com/tobiasgustafsson/immutability-and-python/

Contributors

Tobias Gustafsson https://github.com/tobgu

Christopher Armstrong https://github.com/radix

Anders Hovmöller https://github.com/boxed

Itamar Turner-Trauring https://github.com/itamarst

Jonathan Lange https://github.com/jml

Richard Futrell https://github.com/Futrell

Jakob Hollenstein https://github.com/jkbjh

David Honour https://github.com/foolswood

David R. MacIver https://github.com/DRMacIver

Marcus Ewert https://github.com/sarum90

Jean-Paul Calderone https://github.com/exarkun

Douglas Treadwell https://github.com/douglas-treadwell

Travis Parker https://github.com/teepark

Julian Berman https://github.com/Julian

Contributing

If you experience problems please log them on GitHub. If you want to contribute code, please fork the code and submit a pull request.

CHAPTER 2

API documentation

pmap (*initial={}*, *pre_size=0*)

Create new persistent map, inserts all elements in initial into the newly created map. The optional argument pre_size may be used to specify an initial size of the underlying bucket vector. This may have a positive performance impact in the cases where you know beforehand that a large number of elements will be inserted into the map eventually since it will reduce the number of reallocations required.

```
>>> pmap({'a': 13, 'b': 14})
pmap({'a': 13, 'b': 14})
```

m (***kwargs*)

Creates a new persitent map. Inserts all key value arguments into the newly created map.

```
>>> m(a=13, b=14)
pmap({'a': 13, 'b': 14})
```

class PMap

Persistent map/dict. Tries to follow the same naming conventions as the built in dict where feasible.

Do not instantiate directly, instead use the factory functions *m()* or *pmap()* to create an instance.

Was originally written as a very close copy of the Clojure equivalent but was later rewritten to closer re-assemble the python dict. This means that a sparse vector (a PVector) of buckets is used. The keys are hashed and the elements inserted at position hash % len(bucket_vector). Whenever the map size exceeds 2/3 of the containing vectors size the map is reallocated to a vector of double the size. This is done to avoid excessive hash collisions.

This structure corresponds most closely to the built in dict type and is intended as a replacement. Where the semantics are the same (more or less) the same function names have been used but for some cases it is not possible, for example assignments and deletion of values.

PMap implements the Mapping protocol and is Hashable.

Random access and insert is log32(n) where n is the size of the map.

The following are examples of some common operations on persistent maps

```
>>> m1 = m(a=1, b=3)
>>> m2 = m1.set('c', 3)
>>> m3 = m2.remove('a')
>>> m1
pmap({'a': 1, 'b': 3})
>>> m2
pmap({'a': 1, 'c': 3, 'b': 3})
>>> m3
pmap({'c': 3, 'b': 3})
>>> m3['c']
3
```

discard(*key*)

Return a new PMap without the element specified by key. Returns reference to itself if element is not present.

```
>>> m1 = m(a=1, b=2)
>>> m1.discard('a')
pmap({'b': 2})
>>> m1 is m1.discard('c')
True
```

evolver()

Create a new evolver for this pmap. For a discussion on evolvers in general see the documentation for the pvector evolver.

Create the evolver and perform various mutating updates to it:

```
>>> m1 = m(a=1, b=2)
>>> e = m1.evolver()
>>> e['c'] = 3
>>> len(e)
3
>>> del e['a']
```

The underlying pmap remains the same:

```
>>> m1
pmap({'a': 1, 'b': 2})
```

The changes are kept in the evolver. An updated pmap can be created using the persistent() function on the evolver.

```
>>> m2 = e.persistent()
>>> m2
pmap({'c': 3, 'b': 2})
```

The new pmap will share data with the original pmap in the same way that would have been done if only using operations on the pmap.

get(*k*[, *d*]) → D[k] if k in D, else d. d defaults to None.

remove(*key*)

Return a new PMap without the element specified by key. Raises KeyError if the element is not present.

```
>>> m1 = m(a=1, b=2)
>>> m1.remove('a')
pmap({'b': 2})
```

set (*key*, *val*)
Return a new PMap with key and val inserted.

```
>>> m1 = m(a=1, b=2)
>>> m2 = m1.set('a', 3)
>>> m3 = m1.set('c' ,4)
>>> m1
pmap({'a': 1, 'b': 2})
>>> m2
pmap({'a': 3, 'b': 2})
>>> m3
pmap({'a': 1, 'c': 4, 'b': 2})
```

transform (**transformations*)
Transform arbitrarily complex combinations of PVectors and PMaps. A transformation consists of two parts. One match expression that specifies which elements to transform and one transformation function that performs the actual transformation.

```
>>> from pyrsistent import freeze, ny
>>> news_paper = freeze({'articles': [{'author': 'Sara', 'content': 'A short␣
↪article'},
...                                   {'author': 'Steve', 'content': 'A␣
↪slightly longer article'}],
...                         'weather': {'temperature': '11C', 'wind': '5m/s'}})
>>> short_news = news_paper.transform(['articles', ny, 'content'], lambda c:␣
↪c[:25] + '...' if len(c) > 25 else c)
>>> very_short_news = news_paper.transform(['articles', ny, 'content'],␣
↪lambda c: c[:15] + '...' if len(c) > 15 else c)
>>> very_short_news.articles[0].content
'A short article'
>>> very_short_news.articles[1].content
'A slightly long...'
```

When nothing has been transformed the original data structure is kept

```
>>> short_news is news_paper
True
>>> very_short_news is news_paper
False
>>> very_short_news.articles[0] is news_paper.articles[0]
True
```

update (**maps*)
Return a new PMap with the items in Mappings inserted. If the same key is present in multiple maps the rightmost (last) value is inserted.

```
>>> m1 = m(a=1, b=2)
>>> m1.update(m(a=2, c=3), {'a': 17, 'd': 35})
pmap({'a': 17, 'c': 3, 'b': 2, 'd': 35})
```

update_with (*update_fn*, **maps*)
Return a new PMap with the items in Mappings maps inserted. If the same key is present in multiple maps the values will be merged using merge_fn going from left to right.

```
>>> from operator import add
>>> m1 = m(a=1, b=2)
>>> m1.update_with(add, m(a=2))
pmap({'a': 3, 'b': 2})
```

The reverse behaviour of the regular merge. Keep the leftmost element instead of the rightmost.

```
>>> m1 = m(a=1)
>>> m1.update_with(lambda l, r: l, m(a=2), {'a':3})
pmap({'a': 1})
```

pvector(*[iterable]*)

Create a new persistent vector containing the elements in iterable.

```
>>> v1 = pvector([1, 2, 3])
>>> v1
pvector([1, 2, 3])
```

v(**elements*)

Create a new persistent vector containing all parameters to this function.

```
>>> v1 = v(1, 2, 3)
>>> v1
pvector([1, 2, 3])
```

class PVector

Persistent vector implementation. Meant as a replacement for the cases where you would normally use a Python list.

Do not instantiate directly, instead use the factory functions *v()* and *pvector()* to create an instance.

Heavily influenced by the persistent vector available in Clojure. Initially this was more or less just a port of the Java code for the Clojure vector. It has since been modified and to some extent optimized for usage in Python.

The vector is organized as a trie, any mutating method will return a new vector that contains the changes. No updates are done to the original vector. Structural sharing between vectors are applied where possible to save space and to avoid making complete copies.

This structure corresponds most closely to the built in list type and is intended as a replacement. Where the semantics are the same (more or less) the same function names have been used but for some cases it is not possible, for example assignments.

The PVector implements the Sequence protocol and is Hashable.

Inserts are amortized O(1). Random access is log32(n) where n is the size of the vector.

The following are examples of some common operations on persistent vectors:

```
>>> p = v(1, 2, 3)
>>> p2 = p.append(4)
>>> p3 = p2.extend([5, 6, 7])
>>> p
pvector([1, 2, 3])
>>> p2
pvector([1, 2, 3, 4])
>>> p3
pvector([1, 2, 3, 4, 5, 6, 7])
>>> p3[5]
6
>>> p.set(1, 99)
pvector([1, 99, 3])
>>>
```

append(*val*)

Return a new vector with val appended.

```
>>> v1 = v(1, 2)
>>> v1.append(3)
pvector([1, 2, 3])
```

count(*value*)

Return the number of times that value appears in the vector.

```
>>> v1 = v(1, 4, 3, 4)
>>> v1.count(4)
2
```

delete(*index*, *stop=None*)

Delete a portion of the vector by index or range.

```
>>> v1 = v(1, 2, 3, 4, 5)
>>> v1.delete(1)
pvector([1, 3, 4, 5])
>>> v1.delete(1, 3)
pvector([1, 4, 5])
```

evolver()

Create a new evolver for this pvector. The evolver acts as a mutable view of the vector with "transaction like" semantics. No part of the underlying vector i updated, it is still fully immutable. Furthermore multiple evolvers created from the same pvector do not interfere with each other.

You may want to use an evolver instead of working directly with the pvector in the following cases:

- Multiple updates are done to the same vector and the intermediate results are of no interest. In this case using an evolver may be a more efficient and easier to work with.
- You need to pass a vector into a legacy function or a function that you have no control over which performs in place mutations of lists. In this case pass an evolver instance instead and then create a new pvector from the evolver once the function returns.

The following example illustrates a typical workflow when working with evolvers. It also displays most of the API (which i kept small by design, you should not be tempted to use evolvers in excess ;-)).

Create the evolver and perform various mutating updates to it:

```
>>> v1 = v(1, 2, 3, 4, 5)
>>> e = v1.evolver()
>>> e[1] = 22
>>> _ = e.append(6)
>>> _ = e.extend([7, 8, 9])
>>> e[8] += 1
>>> len(e)
9
```

The underlying pvector remains the same:

```
>>> v1
pvector([1, 2, 3, 4, 5])
```

The changes are kept in the evolver. An updated pvector can be created using the persistent() function on the evolver.

```
>>> v2 = e.persistent()
>>> v2
pvector([1, 22, 3, 4, 5, 6, 7, 8, 10])
```

The new pvector will share data with the original pvector in the same way that would have been done if only using operations on the pvector.

extend(*obj*)
Return a new vector with all values in obj appended to it. Obj may be another PVector or any other Iterable.

```
>>> v1 = v(1, 2, 3)
>>> v1.extend([4, 5])
pvector([1, 2, 3, 4, 5])
```

index(*value*, **args*, ***kwargs*)
Return first index of value. Additional indexes may be supplied to limit the search to a sub range of the vector.

```
>>> v1 = v(1, 2, 3, 4, 3)
>>> v1.index(3)
2
>>> v1.index(3, 3, 5)
4
```

mset(**args*)
Return a new vector with elements in specified positions replaced by values (multi set).

Elements on even positions in the argument list are interpreted as indexes while elements on odd positions are considered values.

```
>>> v1 = v(1, 2, 3)
>>> v1.mset(0, 11, 2, 33)
pvector([11, 2, 33])
```

remove(*value*)
Remove the first occurrence of a value from the vector.

```
>>> v1 = v(1, 2, 3, 2, 1)
>>> v2 = v1.remove(1)
>>> v2
pvector([2, 3, 2, 1])
>>> v2.remove(1)
pvector([2, 3, 2])
```

set(*i*, *val*)
Return a new vector with element at position i replaced with val. The original vector remains unchanged.

Setting a value one step beyond the end of the vector is equal to appending. Setting beyond that will result in an IndexError.

```
>>> v1 = v(1, 2, 3)
>>> v1.set(1, 4)
pvector([1, 4, 3])
>>> v1.set(3, 4)
pvector([1, 2, 3, 4])
>>> v1.set(-1, 4)
pvector([1, 2, 4])
```

transform(**transformations*)
Transform arbitrarily complex combinations of PVectors and PMaps. A transformation consists of two parts. One match expression that specifies which elements to transform and one transformation function that performs the actual transformation.

```
>>> from pyrsistent import freeze, ny
>>> news_paper = freeze({'articles': [{'author': 'Sara', 'content': 'A short␣
↪article'},
...                                   {'author': 'Steve', 'content': 'A␣
↪slightly longer article'}],
...                         'weather': {'temperature': '11C', 'wind': '5m/s'}})
>>> short_news = news_paper.transform(['articles', ny, 'content'], lambda c:␣
↪c[:25] + '...' if len(c) > 25 else c)
>>> very_short_news = news_paper.transform(['articles', ny, 'content'],␣
↪lambda c: c[:15] + '...' if len(c) > 15 else c)
>>> very_short_news.articles[0].content
'A short article'
>>> very_short_news.articles[1].content
'A slightly long...'
```

When nothing has been transformed the original data structure is kept

```
>>> short_news is news_paper
True
>>> very_short_news is news_paper
False
>>> very_short_news.articles[0] is news_paper.articles[0]
True
```

pset (*iterable=()*, *pre_size=8*)

Creates a persistent set from iterable. Optionally takes a sizing parameter equivalent to that used for *pmap()*.

```
>>> s1 = pset([1, 2, 3, 2])
>>> s1
pset([1, 2, 3])
```

s (**elements*)

Create a persistent set.

Takes an arbitrary number of arguments to insert into the new set.

```
>>> s1 = s(1, 2, 3, 2)
>>> s1
pset([1, 2, 3])
```

class PSet

Persistent set implementation. Built on top of the persistent map. The set supports all operations in the Set protocol and is Hashable.

Do not instantiate directly, instead use the factory functions *s()* or *pset()* to create an instance.

Random access and insert is log32(n) where n is the size of the set.

Some examples:

```
>>> s = pset([1, 2, 3, 1])
>>> s2 = s.add(4)
>>> s3 = s2.remove(2)
>>> s
pset([1, 2, 3])
>>> s2
pset([1, 2, 3, 4])
>>> s3
pset([1, 3, 4])
```

add(*element*)
Return a new PSet with element added

```
>>> s1 = s(1, 2)
>>> s1.add(3)
pset([1, 2, 3])
```

discard(*element*)
Return a new PSet with element removed. Returns itself if element is not present.

evolver()
Create a new evolver for this pset. For a discussion on evolvers in general see the documentation for the pvector evolver.

Create the evolver and perform various mutating updates to it:

```
>>> s1 = s(1, 2, 3)
>>> e = s1.evolver()
>>> _ = e.add(4)
>>> len(e)
4
>>> _ = e.remove(1)
```

The underlying pset remains the same:

```
>>> s1
pset([1, 2, 3])
```

The changes are kept in the evolver. An updated pmap can be created using the persistent() function on the evolver.

```
>>> s2 = e.persistent()
>>> s2
pset([2, 3, 4])
```

The new pset will share data with the original pset in the same way that would have been done if only using operations on the pset.

isdisjoint(*other*)
Return True if two sets have a null intersection.

remove(*element*)
Return a new PSet with element removed. Raises KeyError if element is not present.

```
>>> s1 = s(1, 2)
>>> s1.remove(2)
pset([1])
```

update(*iterable*)
Return a new PSet with elements in iterable added

```
>>> s1 = s(1, 2)
>>> s1.update([3, 4, 4])
pset([1, 2, 3, 4])
```

pbag(*elements*)
Convert an iterable to a persistent bag.

Takes an iterable with elements to insert.

```
>>> pbag([1, 2, 3, 2])
pbag([1, 2, 2, 3])
```

b(**elements*)
Construct a persistent bag.

Takes an arbitrary number of arguments to insert into the new persistent bag.

```
>>> b(1, 2, 3, 2)
pbag([1, 2, 2, 3])
```

class PBag(*counts*)
A persistent bag/multiset type.

Requires elements to be hashable, and allows duplicates, but has no ordering. Bags are hashable.

Do not instantiate directly, instead use the factory functions *b()* or *pbag()* to create an instance.

Some examples:

```
>>> s = pbag([1, 2, 3, 1])
>>> s2 = s.add(4)
>>> s3 = s2.remove(1)
>>> s
pbag([1, 1, 2, 3])
>>> s2
pbag([1, 1, 2, 3, 4])
>>> s3
pbag([1, 2, 3, 4])
```

add(*element*)
Add an element to the bag.

```
>>> s = pbag([1])
>>> s2 = s.add(1)
>>> s3 = s.add(2)
>>> s2
pbag([1, 1])
>>> s3
pbag([1, 2])
```

count(*element*)
Return the number of times an element appears.

```
>>> pbag([]).count('non-existent')
0
>>> pbag([1, 1, 2]).count(1)
2
```

remove(*element*)
Remove an element from the bag.

```
>>> s = pbag([1, 1, 2])
>>> s2 = s.remove(1)
>>> s3 = s.remove(2)
>>> s2
pbag([1, 2])
>>> s3
pbag([1, 1])
```

update(*iterable*)
Update bag with all elements in iterable.

```
>>> s = pbag([1])
>>> s.update([1, 2])
pbag([1, 1, 2])
```

plist(*iterable=()*, *reverse=False*)
Creates a new persistent list containing all elements of iterable. Optional parameter reverse specifies if the elements should be inserted in reverse order or not.

```
>>> plist([1, 2, 3])
plist([1, 2, 3])
>>> plist([1, 2, 3], reverse=True)
plist([3, 2, 1])
```

l(**elements*)
Creates a new persistent list containing all arguments.

```
>>> l(1, 2, 3)
plist([1, 2, 3])
```

class PList
Classical Lisp style singly linked list. Adding elements to the head using cons is O(1). Element access is O(k) where k is the position of the element in the list. Taking the length of the list is O(n).

Fully supports the Sequence and Hashable protocols including indexing and slicing but if you need fast random access go for the PVector instead.

Do not instantiate directly, instead use the factory functions *l()* or *plist()* to create an instance.

Some examples:

```
>>> x = plist([1, 2])
>>> y = x.cons(3)
>>> x
plist([1, 2])
>>> y
plist([3, 1, 2])
>>> y.first
3
>>> y.rest == x
True
>>> y[:2]
plist([3, 1])
```

pdeque(*iterable=()*, *maxlen=None*)
Return deque containing the elements of iterable. If maxlen is specified then len(iterable) - maxlen elements are discarded from the left to if len(iterable) > maxlen.

```
>>> pdeque([1, 2, 3])
pdeque([1, 2, 3])
>>> pdeque([1, 2, 3, 4], maxlen=2)
pdeque([3, 4], maxlen=2)
```

dq(**elements*)
Return deque containing all arguments.

```
>>> dq(1, 2, 3)
pdeque([1, 2, 3])
```

class PDeque

Persistent double ended queue (deque). Allows quick appends and pops in both ends. Implemented using two persistent lists.

A maximum length can be specified to create a bounded queue.

Fully supports the Sequence and Hashable protocols including indexing and slicing but if you need fast random access go for the PVector instead.

Do not instantiate directly, instead use the factory functions *dq()* or *pdeque()* to create an instance.

Some examples:

```
>>> x = pdeque([1, 2, 3])
>>> x.left
1
>>> x.right
3
>>> x[0] == x.left
True
>>> x[-1] == x.right
True
>>> x.pop()
pdeque([1, 2])
>>> x.pop() == x[:-1]
True
>>> x.popleft()
pdeque([2, 3])
>>> x.append(4)
pdeque([1, 2, 3, 4])
>>> x.appendleft(4)
pdeque([4, 1, 2, 3])
```

```
>>> y = pdeque([1, 2, 3], maxlen=3)
>>> y.append(4)
pdeque([2, 3, 4], maxlen=3)
>>> y.appendleft(4)
pdeque([4, 1, 2], maxlen=3)
```

append(*elem*)

Return new deque with elem as the rightmost element.

```
>>> pdeque([1, 2]).append(3)
pdeque([1, 2, 3])
```

appendleft(*elem*)

Return new deque with elem as the leftmost element.

```
>>> pdeque([1, 2]).appendleft(3)
pdeque([3, 1, 2])
```

count(*elem*)

Return the number of elements equal to elem present in the queue

```
>>> pdeque([1, 2, 1]).count(1)
2
```

extend(*iterable*)
Return new deque with all elements of iterable appended to the right.

```
>>> pdeque([1, 2]).extend([3, 4])
pdeque([1, 2, 3, 4])
```

extendleft(*iterable*)
Return new deque with all elements of iterable appended to the left.

NB! The elements will be inserted in reverse order compared to the order in the iterable.

```
>>> pdeque([1, 2]).extendleft([3, 4])
pdeque([4, 3, 1, 2])
```

index(*value*) → integer – return first index of value.
Raises ValueError if the value is not present.

left
Leftmost element in dqueue.

maxlen
Maximum length of the queue.

pop(*count=1*)
Return new deque with rightmost element removed. Popping the empty queue will return the empty queue. A optional count can be given to indicate the number of elements to pop. Popping with a negative index is the same as popleft. Executes in amortized O(k) where k is the number of elements to pop.

```
>>> pdeque([1, 2]).pop()
pdeque([1])
>>> pdeque([1, 2]).pop(2)
pdeque([])
>>> pdeque([1, 2]).pop(-1)
pdeque([2])
```

popleft(*count=1*)
Return new deque with leftmost element removed. Otherwise functionally equivalent to pop().

```
>>> pdeque([1, 2]).popleft()
pdeque([2])
```

remove(*elem*)
Return new deque with first element from left equal to elem removed. If no such element is found a ValueError is raised.

```
>>> pdeque([2, 1, 2]).remove(2)
pdeque([1, 2])
```

reverse()
Return reversed deque.

```
>>> pdeque([1, 2, 3]).reverse()
pdeque([3, 2, 1])
```

Also supports the standard python reverse function.

```
>>> reversed(pdeque([1, 2, 3]))
pdeque([3, 2, 1])
```

right
Rightmost element in dqueue.

rotate(*steps*)
Return deque with elements rotated steps steps.

```
>>> x = pdeque([1, 2, 3])
>>> x.rotate(1)
pdeque([3, 1, 2])
>>> x.rotate(-2)
pdeque([3, 1, 2])
```

class CheckedPMap
A CheckedPMap is a PMap which allows specifying type and invariant checks.

```
>>> class IntToFloatMap(CheckedPMap):
...     __key_type__ = int
...     __value_type__ = float
...     __invariant__ = lambda k, v: (int(v) == k, 'Invalid mapping')
...
>>> IntToFloatMap({1: 1.5, 2: 2.25})
IntToFloatMap({1: 1.5, 2: 2.25})
```

class CheckedPVector
A CheckedPVector is a PVector which allows specifying type and invariant checks.

```
>>> class Positives(CheckedPVector):
...     __type__ = (long, int)
...     __invariant__ = lambda n: (n >= 0, 'Negative')
...
>>> Positives([1, 2, 3])
Positives([1, 2, 3])
```

class CheckedPSet
A CheckedPSet is a PSet which allows specifying type and invariant checks.

```
>>> class Positives(CheckedPSet):
...     __type__ = (long, int)
...     __invariant__ = lambda n: (n >= 0, 'Negative')
...
>>> Positives([1, 2, 3])
Positives([1, 2, 3])
```

exception InvariantException(*error_codes=()*, *missing_fields=()*, **args*, ***kwargs*)
Exception raised from a *CheckedType* when invariant tests fail or when a mandatory field is missing.

Contains two fields of interest: invariant_errors, a tuple of error data for the failing invariants missing_fields, a tuple of strings specifying the missing names

exception CheckedKeyTypeError(*source_class*, *expected_types*, *actual_type*, *actual_value*, **args*, ***kwargs*)
Raised when trying to set a value using a key with a type that doesn't match the declared type.

Attributes: source_class – The class of the collection expected_types – Allowed types actual_type – The non matching type actual_value – Value of the variable with the non matching type

exception **CheckedValueTypeError**(*source_class*, *expected_types*, *actual_type*, *actual_value*, **args*, ***kwargs*)

Raised when trying to set a value using a key with a type that doesn't match the declared type.

Attributes: source_class – The class of the collection expected_types – Allowed types actual_type – The non matching type actual_value – Value of the variable with the non matching type

class **CheckedType**

Marker class to enable creation and serialization of checked object graphs.

optional(**typs*)

Convenience function to specify that a value may be of any of the types in type 'typs' or None

class **PRecord**

A PRecord is a PMap with a fixed set of specified fields. Records are declared as python classes inheriting from PRecord. Because it is a PMap it has full support for all Mapping methods such as iteration and element access using subscript notation.

More documentation and examples of PRecord usage is available at https://github.com/tobgu/pyrsistent

classmethod **create**(*kwargs*, *_bypass_factories=False*)

Factory method. Will create a new PRecord of the current type and assign the values specified in kwargs.

evolver()

Returns an evolver of this object.

serialize(*format=None*)

Serialize the current PRecord using custom serializer functions for fields where such have been supplied.

set(**args*, ***kwargs*)

Set a field in the record. This set function differs slightly from that in the PMap class. First of all it accepts key-value pairs. Second it accepts multiple key-value pairs to perform one, atomic, update of multiple fields.

field(*type=()*, *invariant=<function <lambda>>*, *initial=<object object>*, *mandatory=False*, *factory=<function <lambda>>*, *serializer=<function <lambda>>*)

Field specification factory for `PRecord`.

Parameters

- **type** – a type or iterable with types that are allowed for this field
- **invariant** – a function specifying an invariant that must hold for the field
- **initial** – value of field if not specified when instantiating the record
- **mandatory** – boolean specifying if the field is mandatory or not
- **factory** – function called when field is set.
- **serializer** – function that returns a serialized version of the field

pset_field(*item_type*, *optional=False*, *initial=()*)

Create checked `PSet` field.

Parameters

- **item_type** – The required type for the items in the set.
- **optional** – If true, `None` can be used as a value for this field.
- **initial** – Initial value to pass to factory if no value is given for the field.

Returns A `field` containing a `CheckedPSet` of the given type.

pmap_field(*key_type*, *value_type*, *optional=False*, *invariant=<function <lambda>>*)
Create a checked `PMap` field.

Parameters

- **key** – The required type for the keys of the map.
- **value** – The required type for the values of the map.
- **optional** – If true, `None` can be used as a value for this field.
- **invariant** – Pass-through to `field`.

Returns A `field` containing a `CheckedPMap`.

pvector_field(*item_type*, *optional=False*, *initial=()*)
Create checked `PVector` field.

Parameters

- **item_type** – The required type for the items in the vector.
- **optional** – If true, `None` can be used as a value for this field.
- **initial** – Initial value to pass to factory if no value is given for the field.

Returns A `field` containing a `CheckedPVector` of the given type.

class PClass
A PClass is a python class with a fixed set of specified fields. PClasses are declared as python classes inheriting from PClass. It is defined the same way that PRecords are and behaves like a PRecord in all aspects except that it is not a PMap and hence not a collection but rather a plain Python object.

More documentation and examples of PClass usage is available at https://github.com/tobgu/pyrsistent

classmethod create(*kwargs*, *_bypass_factories=False*)
Factory method. Will create a new PClass of the current type and assign the values specified in kwargs.

evolver()
Returns an evolver for this object.

remove(*name*)
Remove attribute given by name from the current instance. Raises AttributeError if the attribute doesn't exist.

serialize(*format=None*)
Serialize the current PClass using custom serializer functions for fields where such have been supplied.

set(**args*, ***kwargs*)
Set a field in the instance. Returns a new instance with the updated value. The original instance remains unmodified. Accepts key-value pairs or single string representing the field name and a value.

```
>>> from pyrsistent import PClass, field
>>> class AClass(PClass):
...     x = field()
...
>>> a = AClass(x=1)
>>> a2 = a.set(x=2)
>>> a3 = a.set('x', 3)
>>> a
AClass(x=1)
>>> a2
AClass(x=2)
```

```
>>> a3
AClass(x=3)
```

transform(**transformations*)

Apply transformations to the currency PClass. For more details on transformations see the documentation for PMap. Transformations on PClasses do not support key matching since the PClass is not a collection. Apart from that the transformations available for other persistent types work as expected.

immutable(*members=''*, *name='Immutable'*, *verbose=False*)

Produces a class that either can be used standalone or as a base class for persistent classes.

This is a thin wrapper around a named tuple.

Constructing a type and using it to instantiate objects:

```
>>> Point = immutable('x, y', name='Point')
>>> p = Point(1, 2)
>>> p2 = p.set(x=3)
>>> p
Point(x=1, y=2)
>>> p2
Point(x=3, y=2)
```

Inheriting from a constructed type. In this case no type name needs to be supplied:

```
>>> class PositivePoint(immutable('x, y')):
...     __slots__ = tuple()
...     def __new__(cls, x, y):
...         if x > 0 and y > 0:
...             return super(PositivePoint, cls).__new__(cls, x, y)
...         raise Exception('Coordinates must be positive!')
...
>>> p = PositivePoint(1, 2)
>>> p.set(x=3)
PositivePoint(x=3, y=2)
>>> p.set(y=-3)
Traceback (most recent call last):
Exception: Coordinates must be positive!
```

The persistent class also supports the notion of frozen members. The value of a frozen member cannot be updated. For example it could be used to implement an ID that should remain the same over time. A frozen member is denoted by a trailing underscore.

```
>>> Point = immutable('x, y, id_', name='Point')
>>> p = Point(1, 2, id_=17)
>>> p.set(x=3)
Point(x=3, y=2, id_=17)
>>> p.set(id_=18)
Traceback (most recent call last):
AttributeError: Cannot set frozen members id_
```

freeze(*o*)

Recursively convert simple Python containers into pyrsistent versions of those containers.

- list is converted to pvector, recursively
- dict is converted to pmap, recursively on values (but not keys)
- set is converted to pset, but not recursively

•tuple is converted to tuple, recursively.

Sets and dict keys are not recursively frozen because they do not contain mutable data by convention. The main exception to this rule is that dict keys and set elements are often instances of mutable objects that support hash-by-id, which this function can't convert anyway.

```
>>> freeze(set([1, 2]))
pset([1, 2])
>>> freeze([1, {'a': 3}])
pvector([1, pmap({'a': 3})])
>>> freeze((1, []))
(1, pvector([]))
```

thaw(*o*)

Recursively convert pyrsistent containers into simple Python containers.

•pvector is converted to list, recursively

•pmap is converted to dict, recursively on values (but not keys)

•pset is converted to set, but not recursively

•tuple is converted to tuple, recursively.

```
>>> from pyrsistent import s, m, v
>>> thaw(s(1, 2))
set([1, 2])
>>> thaw(v(1, m(a=3)))
[1, {'a': 3}]
>>> thaw((1, v()))
(1, [])
```

mutant(*fn*)

Convenience decorator to isolate mutation to within the decorated function (with respect to the input arguments).

All arguments to the decorated function will be frozen so that they are guaranteed not to change. The return value is also frozen.

get_in(*keys*, *coll*, *default=None*, *no_default=False*)

NB: This is a straight copy of the get_in implementation found in the toolz library (https://github.com/pytoolz/toolz/). It works with persistent data structures as well as the corresponding datastructures from the stdlib.

Returns coll[i0][i1]...[iX] where [i0, i1, ..., iX]==keys.

If coll[i0][i1]...[iX] cannot be found, returns `default`, unless `no_default` is specified, then it raises KeyError or IndexError.

`get_in` is a generalization of `operator.getitem` for nested data structures such as dictionaries and lists. >>> from pyrsistent import freeze >>> transaction = freeze({'name': 'Alice', ... 'purchase': {'items': ['Apple', 'Orange'], ... 'costs': [0.50, 1.25]}, ... 'credit card': '5555-1234-1234-1234'}) >>> get_in(['purchase', 'items', 0], transaction) 'Apple' >>> get_in(['name'], transaction) 'Alice' >>> get_in(['purchase', 'total'], transaction) >>> get_in(['purchase', 'items', 'apple'], transaction) >>> get_in(['purchase', 'items', 10], transaction) >>> get_in(['purchase', 'total'], transaction, 0) 0 >>> get_in(['y'], {}, no_default=True) Traceback (most recent call last):

...

KeyError: 'y'

inc(*x*)
Add one to the current value

discard(*evolver*, *key*)
Discard the element and returns a structure without the discarded elements

rex(*expr*)
Regular expression matcher to use together with transform functions

ny(*_*)
Matcher that matches any value

CHAPTER 3

Revision history

v0.13.0, 2017-09-01

- Fix #113, Skip field factories when loading pickled objects. There is a minor backwards incompatibilty in the behaviour because of this. Thanks @teepark for fi this!
- Fix #116, negative indexing for pdeques. Thanks @Julian for this!

v0.12.3, 2017-06-04

- Fix #83, make it possible to use Python 3 enums as field type without having to wrap it in a list or tuple. Thanks @douglas-treadwell for this!

v0.12.2, 2017-05-30

- Fix #108, now possible to use the values in predicates to transform. Thanks @exarkus for this!
- Fix #107, support multiple level of __invariant__ inheritance. Thanks @exarkus for this!

v0.12.1, 2017-02-26

- Fix #97, initialize CheckedPVector from iterator-
- Fix #97, cache hash value on PMap. Thanks @sarum90 for this!

v0.12.0, 2017-01-06

- Fix #87, add function get_in() for access to elements in deeply nested structures.
- Fix #91, add method update() to pset and pbag.
- Fix #92, incorrect discard of elements in transform on pvector
- This is a release candidate for 1.0 as I now consider pyrsistent fairly stable.

v0.11.13, 2016-04-03

- Fix #84, pvector segfault in CPython 3 when repr of contained object raises Exception.
- Update README to cover for issue described in #83.

v0.11.12, 2016-02-06

- Minor modifications of tests to allow testing as requested in #79 and #80.
- Also run CI tests under python 3.5

v0.11.11, 2016-01-31

- #78, include tests in pypi dist.

v0.11.10, 2015-12-27, NOTE! This release contains a backwards incompatible change

despite only stepping the patch version number. See below.

- Implement #74, attribute access on PClass evolver
- Implement #75, lazily evaluated invariant messages by providing a callable with no arguments.
- Initial values on fields can now be evaluated on object creation by providing a callable with no arguments.

  NOTE! If you previously had callables as initial values this change means that those will be called upon object creation which may not be what you want. As a temporary workaround a callable returning a callable can be used. This feature and the concept of initial values will likely change slightly in the future. See #77 and and #76 for more information.

v0.11.9, 2015-11-01

- Added PVector.remove(), thanks @radix for initiating this!

v0.11.8, 2015-10-18

- Fix #66, UnicodeDecodeError when doing pip install in environments with ascii encoding as default. Thanks @foolswood!
- Implement support for multiple types in pmap_field(), pvector_field() and pset_field(). Thanks @itamarst!

v0.11.7, 2015-10-03

- Fix #52, occasional SEGFAULTs due to misplaced call to PyObject_GC_Track. Thanks @jkbjh for this!
- Fix #42, complete support for delete. Now also on the C-implementation of the PVectorEvolver. Thanks @itamarst for contributing a whole bunch of Hypothesis test cases covering the evolver operations!

v0.11.6, 2015-09-30

- Add +, -, & and | operations to PBag. Thanks @Futrell for this!

v0.11.5, 2015-09-29

- Fix bug introduced in 0.11.4 that prevented multi level inheritance from PClass.
- Make PClassMeta public for friendlier subclassing

v0.11.4, 2015-09-28

- Fix #59, make it possible to create weakrefs to all collection types. Thanks @itamarst for reporting it.
- Fix #58, add __str__ to InvariantException. Thanks @tomprince for reporting it.

v0.11.3, 2015-09-15

- Fix #57, support pickling of PClasses and PRecords using pmap_field, pvector_field, and pset_field. Thanks @radix for reporting this and submitting a fix for it!

v0.11.2, 2015-09-09

- Fix bug causing potential element loss when reallocating PMap. Thanks to @jml for finding this and submitting a PR with a fix!
- Removed python 3.2 test build from Travis. There is nothing breaking 3.2 compatibility in this release but there will be no effort moving forward to keep the 3.2 compatibility.

v0.11.1, 2015-08-24

- Fix #51, PClass.set() broken when used with string+value argument.
- #50, make it possible to specify more than one assertion in an invariant
- #48, make it possible to make recursive type references by using a string as type specification.

v0.11.0, 2015-07-11

- #42, delete() function added to PVector to allow deletion of elements by index and range. Will perform a full copy of the vector, no structural sharing. Thanks @radix for helping out with this one!
- Fix #39, explicitly disallow ordering for PMap and PBag, Python 3 style
- Fix #37, PMap.values()/keys()/items() now returns PVectors instead of lists

v0.10.3, 2015-06-13

- Fix #40, make it possible to disable the C extension by setting the PYRSISTENT_NO_C_EXTENSION environment variable.

v0.10.2, 2015-06-07

- Fix #38, construction from serialized object for pvector/pset/pmap fields.

v0.10.1, 2015-04-27

- Fix broken README.rst

v10.0.0, 2015-04-27

- New type PClass, a persistent version of a Python object. Related to issues #30 and #32. Thanks @exarkun and @radix for input on this one!
- Rename PRecordTypeError -> PTypeError, it is now also raised by PClass
- New convenience functions, pvector_field, pmap_field and pset_field to create PRecord/PClass fields for checked collections. Issues #26 and #36. Thanks to @itamarst for this!
- Removed deprecated function set_in() on PMap and PVector.
- Removed deprecated factory function pclass.
- Major internal restructuring breaking pyrsistent.py into multiple files. This should not affect those only using the public interface but if you experience problems please let me know.

v0.9.4, 2015-04-20

- Fix #34, PVector now compares against built in list type

v0.9.3, 2015-04-06

- Rename pclass back to immutable and deprecate the usage of the pclass function. PClass will be used by a new, different type in upcoming releases.
- Documentation strings for the exceptions introduced in 0.9.2.

v0.9.2, 2015-04-03

- More informative type errors from checked types, issue #30
- Support multiple optional types, issue #28

v0.9.1, 2015-02-25

- Multi level serialization for checked types

v0.9.0, 2015-02-25, Lots of new stuff in this release!

- Checked types, checked versions of PVector, PMap, PSet that support type and invariant specification. Currently lacking proper documentation but I'm working on it.
- set_in() on PVector and PMap are now deprecated and will be removed in the next release. Use transform() instead. set_in() has been updated to use transform() for this release this means that some corner error cases behave slightly different than before.
- Refactoring of the PVector to unify the type. Should not have any user impact as long as only the public interface of pyrsistent has been used. PVector is now an abstract base class with which the different implementations are registered.
- Evolvers have been updated to return themselves for evolving operations to allow function chaining.
- Richer exception messages for KeyErrors and IndexErrors specifying the key/index that caused the failure. Thanks @radix for this.
- Missing attribute on PMaps when accessing with dot-notation now raises an AttributeError instead of a KeyError. Issue #21.
- New function decorator @mutant that freezes all input arguments to a function and the return value.
- Add __version__ to pyrsistent.py. Issue #23.
- Fix pickling for pset. Issue #24.

v0.8.0, 2015-01-21

- New type PRecord. Subtype of PMap that allows explicit, declarative field specification. Thanks @boxed for inspiration!
- Efficient transformations of arbitrary complexity on PMap and PVector. Thanks @boxed for inspiration!
- Breaking change to the evolver interface. What used to be .pvector(), .pmap() and .pset() on the different evolvers has now been unified so that all evolvers have one method .persistent() to produce the persistent counterpart. Sorry for any inconvenience.
- Removed the tests directory from the package.
- PMap and PSet now contains a copy-function to closer mimic the interface of the dict and set. These functions will simply return a reference to self.
- Removed deprecated alias 'immutable' from pclass.

v0.7.1, 2015-01-17

- Fixes #14 where a file executed (unexpectedly) during installation was not python 3 compatible.

v0.7.0, 2015-01-04, No 1.0, instead a bunch of new stuff and one API breaking change to PMap.remove().

- Evolvers for pvector, pmap and pset to allow simple and efficient updates of multiple elements in the collection. See the documentation for a closer description.
- New method mset on pvector to update multiple values in one operation
- Remove deprecated methods merge and merge_with on PMap
- Change behavior of PMap.remove, it will now raise a KeyError if the element is not present. New method PMap.discard will instead return the original pmap if the element is not present. This aligns the PMap with how things are done in the PSet and is closer to the behavior of the built in counterparts.

v0.6.3, 2014-11-27

- Python 2.6 support, thanks @wrmsr!
- PMap.merge/merge_with renamed to update/update_with. merge/merge_with remains but will be removed for 1.0.

- This is a release candidate for 1.0! Please be aware that PMap.merge/merge_with and immutable() will be removed for 1.0.

v0.6.2, 2014-11-03

- Fix typo causing the pure python vector to be used even if the C implementation was available. Thanks @zerc for finding it!

v0.6.1, 2014-10-31

- Renamed 'immutable' to 'pclass' for consistency but left immutable for compatibility.

v0.6.0, 2014-10-25

- New data structure, persistent linked list
- New data structure, persistent double ended queue

v0.5.0, 2014-09-24

- New data structure, persistent bag / multiset
- New functions freeze and thaw to recursively convert between python built in data types and corresponding pyrsistent data types.
- All data structures can now be pickled
- New function merge_in on persistent map which allows a user supplied function to implement the merge strategy.

v0.4.0, 2014-09-20

- Full Python 3 support.
- Immutable object implemented.
- Bug fixes in PVector.__repr__() and PMap.__hash__() and index check of PVector.
- Repr changed to be fully cut and paste compatible
- Changed assoc() -> set(), assoc_in() -> set_in(), massoc() -> mset(). Sorry for the API breaking change but I think those names are more pythonic.
- Improved documentation.

v0.3.1, 2014-06-29

- assoc() on PSet renamed back to add()

v0.3.0, 2014-06-28

- Full Sequence protocol support for PVector
- Full Mapping protocol support for PMap
- Full Set protocol support for PSet
- assoc_in() support for both PMap and PVector
- merge() support for PMap
- Performance improvements to the PVector C extension speed up allocation

v0.2.1, 2014-06-21

- Supply the tests with the distribution

v0.2.0, 2014-06-21

- New C extension with an optimized version of the persistent vector

- Updated API slightly

v0.1.0, 2013-11-10

- Initial release.

CHAPTER 4

TODO (in no particular order)

- Versioned data structure where the different versions can be accessed by index?
- Ordered sets and maps
- A good performance measurement suite

CHAPTER 5

Indices and tables

- genindex
- modindex
- search

Python Module Index

p

Index

R

S

T

U

V