Anything that can be viewed as a list, regardless of its actual implementation.

Supporting first, rest, cons, as described in clojure.lang. ISeq.

(seq coll)
(next aseq) ;; aka (seq (rest aseq))

... seqs.

In an inner class of the collection. This leads to mangled

names like:

clojure.lang.SomeCollection\$Seg

The key/value pairs are the elements.

Use sorted sets and sorted maps.

(sorted-set & elements)

(sorted-map & elements)

They are like cons and concat, but they add the elements in the position most efficient for the underlying representation.

```
(conj coll element & elements)
(into to-coll from-coll)
```

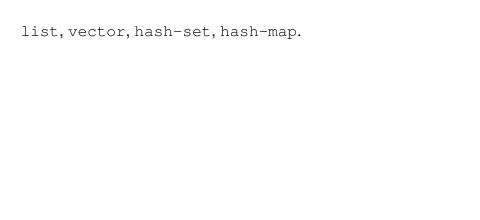
... immutable ... lazy.

(interpose separator coll)
user=> (apply str (interpose ", " ["a" "b" "c"]))
"a, b, c"

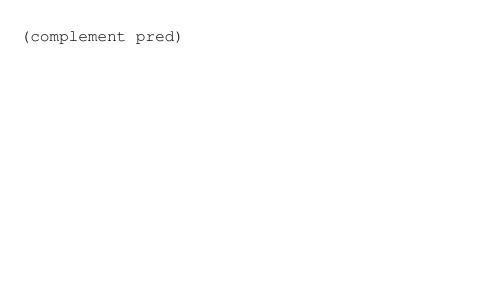
clojure.contrib.str-util's str-join wraps the common apply str pattern.

vec and set take a single collection argument, hash-set

and vector take variable elements.



(take-while pred coll)
(drop-while pred coll)
(split-at index coll)
(split-with pred coll)



(every? pred coll)
(some pred coll)
(not-every? pred coll)
(not-any? pred coll)

```
(map f coll)
(reduce f val? coll)
(sort comp? coll)
(sort-by keyfn comp? coll) ; keyfn is for *getting* the keys
```

(for [binding-form coll-expr filter-expr? ...] expr) There can be may binding-form/coll-expr pairs in a row.

Seq comprehensions are a macro.

- -: when expr which is much like a standard filter/guard.
- -: while expr which stops the comprehension as soon as the predicate fails.

•	stores result of traversal and returns doesn't store, returns nil

- Java collections, arrays, and strings
- Regexps - File hierarchies
- Streams
- XML trees
- Database results

(peek coll)
(peek coll)

peek returns the first element, pop is like rest but throws an exception on an empty sequence.

peek which returns the last element, and pop which returns the "init" of a vector.

```
(get vector index)
(vector index)
(subvec vector start end?)
```

take/drop work on any sequence, but subvec is much

faster for vectors.

(keys map)
(vals map)
(get map key not-found?)
(a-map element); test for membership
(a-keyword map); test for membership

You can't know if a result of nil indicates they key was not in the map or if it was present, mapped to nil.

You can get around the problem with:

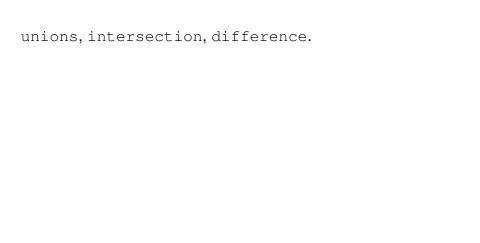
(contains? map key)
(get map key not-found?)

(assoc map key val & more-kvs)
(dissoc map key & more-keys)
(select-keys map key-seq)
(merge map1 map2); map2 wins if both keys exist

(merge-with merge-fn & maps)

... imported to be used unqualified.

(use 'clojure.set)



There are two important correspondences between relational algebra, databases, and the clojure type system.

relation = table = set-like tuple = row = map-like

So in clojure you might have a set of maps (i.e., a relation).

Straight sets: (select pred set)

```
Relations (sets of maps):
(rename relation rename-map)
(project relation keys)
(join rel1 rel2 keymap?)
```