Since the introduction of the `append` built-in, most of the functionality of the `container/vector` package, which was removed in Go 1, can be replicated using `append` and `copy`.

Here are the vector methods and their slice-manipulation analogues:

AppendVector

```
a = append(a, b...)
```

Copy

```
b := make([]T, len(a))
copy(b, a)

// These two are often a little slower than the above one,
// but they would be more efficient if there are more
// elements to be appended to b after copying.
b = append([]T(nil), a...)
b = append(a[:0:0], a...)

// This one-line implementation is equivalent to the above
// two-line make+copy implementation logically. But it is
// actually a bit slower (as of Go toolchain v1.16).
b = append(make([]T, 0, len(a)), a...)
```

Cut

```
a = append(a[:i], a[j:]...)
```

Delete

```
a = append(a[:i], a[i+1:]...)
// or
a = a[:i+copy(a[i:], a[i+1:])]
```

Delete without preserving order

```
a[i] = a[len(a)-1]
a = a[:len(a)-1]
```

NOTE If the type of the element is a *pointer* or a struct with pointer fields, which need to be garbage collected, the above implementations of `Cut` and `Delete` have a potential *memory leak* problem: some elements with values are still referenced by slice `a` and thus can not be collected. The following code can fix this problem: > **Cut**

```
copy(a[i:], a[j:])
for k, n := len(a)-j+i, len(a); k < n; k++ {
```

```
    a[k] = nil // or the zero value of T
}
a = a[:len(a)-j+i]
```

Delete

```
copy(a[i:], a[i+1:])
a[len(a)-1] = nil // or the zero value of T
a = a[:len(a)-1]
```

Delete without preserving order

```
a[i] = a[len(a)-1]
a[len(a)-1] = nil
a = a[:len(a)-1]
```

Expand Insert `n` elements at position `i`:

```
a = append(a[:i], append(make([]T, n), a[i:]...)...)
```

Extend Append `n` elements:

```
a = append(a, make([]T, n)...)
```

Extend Capacity Make sure there is space to append `n` elements without re-allocating:

```
if cap(a)-len(a) < n {
    a = append(make([]T, 0, len(a)+n), a...)
}
```

Filter (in place)

```
n := 0
for _, x := range a {
    if keep(x) {
        a[n] = x
        n++
    }
}
a = a[:n]
```

Insert

```
a = append(a[:i], append([]T{x}, a[i:]...)...)
```

NOTE: The second `append` creates a new slice with its own underlying storage and copies elements in `a[i:]` to that slice, and these elements are then copied back to slice `a` (by the first `append`). The creation of the new slice (and thus memory garbage) and the second copy can be avoided by using an alternative way: > **Insert**

```
s = append(s, 0 /* use the zero value of the element type */)
copy(s[i+1:], s[i:])
s[i] = x
```

InsertVector

```
a = append(a[:i], append(b, a[i:]...)...)

// The above one-line way copies a[i:] twice and
// allocates at least once.
// The following verbose way only copies elements
// in a[i:] once and allocates at most once.
// But, as of Go toolchain 1.16, due to lacking of
// optimizations to avoid elements clearing in the
// "make" call, the verbose way is not always faster.
//
// Future compiler optimizations might implement
// both in the most efficient ways.
//
// Assume element type is int.
func Insert(s []int, k int, vs ...int) []int {
    if n := len(s) + len(vs); n <= cap(s) {
        s2 := s[:n]
        copy(s2[k+len(vs):], s[k:])
        copy(s2[k:], vs)
        return s2
    }
    s2 := make([]int, len(s) + len(vs))
    copy(s2, s[:k])
    copy(s2[k:], vs)
    copy(s2[k+len(vs):], s[k:])
    return s2
}

a = Insert(a, i, b...)
```

Push

```
a = append(a, x)
```

Pop

```
x, a = a[len(a)-1], a[:len(a)-1]
```

Push Front/Unshift

```
a = append([]T{x}, a...)
```

Pop Front/Shift

```
x, a = a[0], a[1:]
```

Additional Tricks

Filtering without allocating

This trick uses the fact that a slice shares the same backing array and capacity as the original, so the storage is reused for the filtered slice. Of course, the original contents are modified.

```
b := a[:0]
for _, x := range a {
    if f(x) {
        b = append(b, x)
    }
}
```

For elements which must be garbage collected, the following code can be included afterwards:

```
for i := len(b); i < len(a); i++ {
    a[i] = nil // or the zero value of T
}
```

Reversing

To replace the contents of a slice with the same elements but in reverse order:

```
for i := len(a)/2-1; i >= 0; i-- {
    opp := len(a)-1-i
    a[i], a[opp] = a[opp], a[i]
}
```

The same thing, except with two indices:

```
for left, right := 0, len(a)-1; left < right; left, right = left+1, right-1 {
    a[left], a[right] = a[right], a[left]
}
```

Shuffling

Fisher–Yates algorithm:

> Since go1.10, this is available at math/rand.Shuffle

```go
for i := len(a) - 1; i > 0; i-- {
    j := rand.Intn(i + 1)
    a[i], a[j] = a[j], a[i]
}
```

Batching with minimal allocation

Useful if you want to do batch processing on large slices.

```go
actions := []int{0, 1, 2, 3, 4, 5, 6, 7, 8, 9}
batchSize := 3
batches := make([][]int, 0, (len(actions) + batchSize - 1) / batchSize)

for batchSize < len(actions) {
    actions, batches = actions[batchSize:], append(batches, actions[0:batchSize:batchSize])
}
batches = append(batches, actions)
```

Yields the following:

```
[[0 1 2] [3 4 5] [6 7 8] [9]]
```

In-place deduplicate (comparable)

```go
import "sort"

in := []int{3,2,1,4,3,2,1,4,1} // any item can be sorted
sort.Ints(in)
j := 0
for i := 1; i < len(in); i++ {
    if in[j] == in[i] {
        continue
    }
    j++
    // preserve the original data
    // in[i], in[j] = in[j], in[i]
    // only set what is required
    in[j] = in[i]
}
result := in[:j+1]
fmt.Println(result) // [1 2 3 4]
```

Move to front, or prepend if not present, in place if possible.

```go
// moveToFront moves needle to the front of haystack, in place if possible.
func moveToFront(needle string, haystack []string) []string {
    if len(haystack) != 0 && haystack[0] == needle {
        return haystack
```

```go
    }
    prev := needle
    for i, elem := range haystack {
        switch {
        case i == 0:
            haystack[0] = needle
            prev = elem
        case elem == needle:
            haystack[i] = prev
            return haystack
        default:
            haystack[i] = prev
            prev = elem
        }
    }
    return append(haystack, prev)
}

haystack := []string{"a", "b", "c", "d", "e"} // [a b c d e]
haystack = moveToFront("c", haystack)         // [c a b d e]
haystack = moveToFront("f", haystack)         // [f c a b d e]
```

Sliding Window

```go
func slidingWindow(size int, input []int) [][]int {
    // returns the input slice as the first element
    if len(input) <= size {
        return [][]int{input}
    }

    // allocate slice at the precise size we need
    r := make([][]int, 0, len(input)-size+1)

    for i, j := 0, size; j <= len(input); i, j = i+1, j+1 {
        r = append(r, input[i:j])
    }

    return r
}
```