The `set!` function is a general way to change a defined value.

For instance, you might want to use a global variable to store your grocery list:

```
(define groceries
  (list "carrots" "avocados" "hummus"))
```

And you could add items to your grocery list by making a new list every time, e.g.,

```
(define groceries-v2
  (cons "bread" groceries))
```

But that’s not really what we want; we want to actually change the list. The best approach is to write a function using `set!`:

```
(define add-grocery!
  (lambda (item)
    (set! groceries (cons item groceries))))
```

When this is called, the global variables `groceries` is defined to be a new list made by `cons`-ing item onto the existing list.

```
> (add-grocery! "chocolate")
> groceries
("chocolate" "carrots" "avocados" "hummus")
```
**set! vs define**

We can use define at the top-level (e.g., in the Interactions Window) to update a previously defined value, e.g.,

```
> (define x "one thing")
> x
"one thing"
> (define x "another")
> x
"another"
```

But it won't work correctly in a function. If we replace `set!` with `define` in `add-grocery!`, we'll get an error.

To change values that are parts of data structures, we need more specific mutator functions.

**Destructive programming and vectors**

A **vector** is Scheme's version of the **array** data type found in many programming languages, which is an ordered collection of elements that can be accessed by an integer index.

Usually all elements of an array have the same type, though in Scheme's vectors they don't need to.

In most languages, the actual values are stored in the array. In Scheme, pointers are stored, which means that the contents of a vector can be of any size without needing to make a bigger vector.
Making vectors

Unlike lists, vectors are not constructed incrementally; they are created with a fixed length in one shot.

`(#(item1 item2 ...)`) creates an immutable vector. Like `quote` notation for lists, the arguments to `#` are shielded from evaluation.

```
> '(2 3 4 (+ 2 3))
(2 3 4 (+ 2 3))
> #(2 3 4 (+ 2 3))
#(2 3 4 (+ 2 3))
```

Making vectors

Unlike lists, vectors are not constructed incrementally; they are created with a fixed length in one shot.

The `vector` function is like the `list` function, letting you specify the items, which are evaluated:

```
> (list 2 3 4 (+ 2 3))
(2 3 4 5)
> (vector 2 3 4 (+ 2 3))
#(2 3 4 5)
```

Making vectors

Unlike lists, vectors are not constructed incrementally; they are created with a fixed length in one shot.

The `make-vector` function returns a vector of the specified length. It can also optionally take a second input specifying the initial contents of the vector.

```
> (make-vector 15)
#(0 0 0 0 0 0 0 0 0 0 0 0 0 0 0)
> (make-vector 15 'a)
#(a a a a a a a a a a a a a a a a a a a a)
```

We can access vector elements by their index using `vector-ref`. This is like the `list-ref` function we've seen, but `list-ref` needs to walk through the list to find an element:

```
> (define v1 (vector 'tom 'dick 'harry))
> (vector-ref v1 1)
dick
```
We can destructively modify vector elements by their index using `vector-set!`.

```scheme
> (define v1 (vector 'tom 'dick 'harry))
> (vector-set! v1 1 'jane)
> v1
#(tom jane harry)
```

Lists vs vectors

Since lists and vectors have essentially the same functionality, why do we have both? Efficiency!

Under the hood, lists are linked sequences of cons cells, with the `cdr` of each cons cell holding the `address` of the next cell in the sequence. These cons cells might be scattered all over the computer’s memory. In contrast, the entries in a vector are allocated all at once and are stored in one contiguous block of memory. This makes for fast access times to arbitrary vector elements.

Lists are good for doing list-based recursion, especially if we’re not creating lots of new cons cells in the process.

Constant-time (i.e., fast) operations for lists are `first`, `rest`, `cons`, and `empty`?

Slow operations are `length`, `list-ref`, and `append`.

Vectors, on the other hand, are good for other purposes in which we jump from place to place in the vector, erasing and re-writing the contents of entries in any order.

Constant-time (i.e., fast) operations for vectors are `vector-ref`, `vector-set!`, and `vector-length`.

There are no equivalents to `first`, `rest`, or `cons` for vectors. Instead of doing recursion until the vector is empty, you need to do recursion over a position number, often ending when the position number is equal to the length of the vector.

```scheme
;;;; VECTOR-SWAP!
;;;; ------------
;;;; INPUTS: VEC, a vector
;;;; I, J, numerical indices
;;;; OUTPUT: void
;;;; SIDE-EFFECT: Swaps the values stored at indices I and J in the vector VEC.
(define vector-swap!
  (lambda (vec i j)
    ;; First, store Ith entry in a local variable, TMP
    (let ((tmp (vector-ref vec i)))
      ;; Then, change Ith entry
      (vector-set! vec i (vector-ref vec j))
      ;; Then, change Jth entry
      (vector-set! vec j tmp)))))

> (define v (vector 'a 'b 'c))
> (vector-swap! v 0 2)
> v
#(c b a)
```
When we define a structure with `define-struct`, it does even more than we learned previously. In addition to defining a constructor, type checker predicate, and accessors for each field, it also defines a mutator function for each field.

For example, if we define a structure `posn` for an x, y position:

```
(define-struct posn (x y))
```

Then Scheme defines `set-posn-x!` and `set-posn-y!` for us:
```
> (define p (make-posn 10 20))
> (set-posn-x! p 100)
> (posn-x p)
100
```
What if we wanted to [travel back in time and] hold everyone back? I.e., everyone’s graduating a year later.

> (map person-grad-class *people*)
(1917 1928 1934)
> (for-each (lambda (x)
   (set-person-grad-class! x
   (add1 (person-grad-class x)))))
*people*
> (map person-grad-class *people*)
(1918 1929 1935)
The general iterative special forms (loop and dotimes) can be used for lists.

But there's also a special form designed just for going through the items in a list: dolist

How could we iteratively compute the length of a list?

> (list-length '(a b c d e))
(a b c d e 5)

You can also think of dolist as a more convenient version of for-each when we don't have an existing function to use:

(for-each (lambda (x)
            (printf "~A~%" x))
          '(1 2 3))

is equivalent to

(dolist (x '(1 2 3))
     (printf "~A~%" x))
How could we use iteration to print the contents a vector?

```
> (print-vector #(a b c))
0: a
1: b
2: c
```

---

How could we use iteration to print the contents of a vector in reverse order?

```
> (print-vector-in-reverse #(a b c))
2: c
1: b
0: a
```
;; PRINT-VECTOR-IN-REVERSE
;; ----------------------------------------
;; INPUTS: VEC, a vector
;; OUTPUT: None
;; SIDE EFFECT: Displays the contents of the
;; vector in the interactions window in *reverse* order.
(define print-vector-in-reverse
  (lambda (vec)
    ;; Store the LENGTH in a local variable,
    ;; just for convenience
    (let ((len (vector-length vec)))
      (dotimes (i len)
        ;; REV-INDY = index in reverse order
        (let ((rev-i (- len 1 i)))
          (printf "~A: ~A~%
                  (vector-ref vec rev-i)))))
    > (print-vector-in-reverse #(a b c))
    2: c
    1: b
    0: a

Acknowledgments

This lecture incorporates material from:
Simon Ellis
Luke Hunsberger
Jennifer Walter