Any questions on lists?

```
>>> lst = [4, "abc", [7, 8], 5]
```

Len + indexing

```
>>> len(lst)
4
>>> lst[0]
4
>>> lst[2]
[7, 8]
```

`len`: number of elements
Indexing: start at 0

Slices

```
>>> lst[0:2]
[4, 'abc']
>>> lst[1:]
['abc', [7, 8], 5]
>>> lst[:1]
[4]
```

Slices: `lst[start:stop]`
Goes from `start` (inclusive)
to `stop` (exclusive)
Default start: `0`,
Default stop: `len(lst)`

Loops

```
>>> for elem in lst:
...     print(elem)
...
4
abc
[7, 8]
5
```

```
for elem in list:
    do_something(elem)
```

Discussion 4:
Data Abstraction & Sequences

Caroline Lemieux (clemieux@berkeley.edu)

February 21st, 2018

Administrativa

Homeworks

HW 3 due today (2/21)

Projects

Maps Project released and due Thursday 2/28
Optional Hog strategy contest ends Friday 2/22.
Hog composition scores on OK

Other

Caroline’s website on the cs61a website! Easy links!

List Comprehensions

```
[map-expr for name in iter-expr if filter-expr]
```

Input: an existing list → **(Optional) Filter**: get rid of elements that don't fulfill condition → **Map:** Applies map expression to each element → **Output:** a new list

```
[x*x for x in [1, 2, 3, 4, 5] if x % 2 == 1]
```

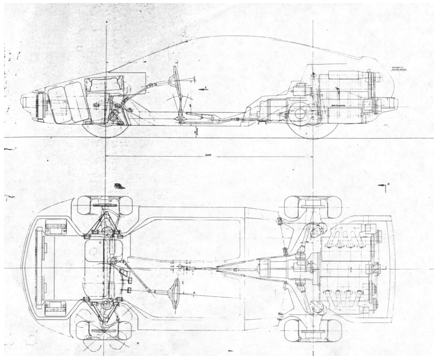
[1, 2, 3, 4, 5] → Keeps elements where x % 2 == 1 **[1, 3, 5]** → Squares each element → **[1, 9, 25]**

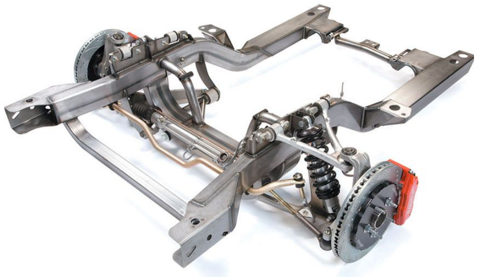
Data Abstraction

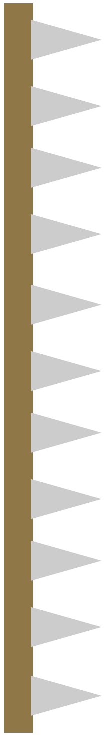
Car abstraction

abstraction barrier

What the car manufacturer sees:







What the end-user sees/uses:

Discussion Section ADT

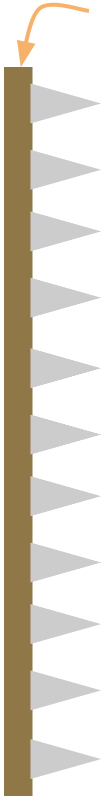
The implementation:

```
def make_discussion(ta, time, students):
    return [name, time, students]

def get_ta(disc):
    return disc[0]

def get_time(disc):
    return disc[1]

def get_students(disc):
    return disc[2]
```



abstraction barrier

What the end-user sees/uses:

Constructor: make_discussion

Selectors:
get_ta
get_time
get_students

Note: the body (implementation of these functions) is hidden!

Worksheet time

Attendance

links.cs61a.org/caro-disc

Recursion on Lists

Let’s go through a familiar problem...

Write `make_zipper`, which, given a list of functions `[f1, f2, f3]` returns a function like

`lambda x: f1(f2(f3(x)))`

E.g. `make_zipper([square, double])` → `lambda x: square(double(x))`

```
>>> make_zipper([])
    lambda x: x
>>> make_zipper([f1])
    lambda x: f1(x)
```

Let’s go through a familiar problem…

Write `make_zipper`, which, given a list of functions `[f1, f2, f3]` returns a function like

```
lambda x: f1(f2(f3(x)))
```

E.g. `make_zipper([square, double])` → `lambda x: square(double(x))`

```
>>> make_zipper([])
    lambda x: x
>>> make_zipper([f1])
    lambda x: f1(x)
```

How can we fit `lambda x : x` here?

Let's go through a familiar problem...

Write `make_zipper`, which, given a list of functions `[f1, f2, f3]` returns a function like

```
lambda x: f1(f2(f3(x)))
```

E.g. `make_zipper([square, double])` → `lambda x: square(double(x))`

```
>>> make_zipper([])

    lambda x: x

>>> make_zipper([f1])

    lambda x: f1(x)

    lambda x: f1(lambda x : x (x))
```

How can we fit `lambda x : x` here?

These are the same!

Skeleton...

```
def make_zipper(fn_lst):
    if lst == []:
        return ________________
    else:
        first_fn = ________________
        rest_of_fns = ________________
        return ________________
```

Answer

```
def make_zipper(fn_lst):
    if lst == []:
        return lambda x : x
    else:
        first_fn = lst[0]
        rest_of_fns = lst[1:]
        return lambda x: first_fn(make_zipper(rest_of_fns)(x))
```

General format of list-recursive questions

```
def recurse_on_list(lst):
    if lst == []:
        return <base_case_value>
    else:
        first = lst[0]
        rest = lst[1:]
        return <combine>(first, recurse_on_list(rest))
```

Typical base case: `lst == []` (same as `not lst` -- why?)

Typical recursive case: Use the first element `lst[0]`, recurse on the rest of the list `lst[1:]`