~~def~~ 61A Cheat Sheet

Richard Zhu

General Tips FOCUS - Check unsure points

* Pay attention to return types
* * Read question descs carefully
  * Read the fucking DOCTESTS
* Follow pointers carefully
* Pay attention to mutations
* be mindful of checks; cover all cases
* recognize nonlocal
  i nonlocal
  REASSIGNMENT

☆ BE CAREFUL on ENV DIAGRAMS with mutated functions
L original operator is bound by the order at time of each step

Function Call order:

1. eval operator
2. eval operands (L→R)
3. apply operator to operands

be careful w/ prints:
print(x) returns None

Macro evaluation order

* usually the operands will be unquoted in implementation

1. eval operator
2. eval the body of the macro procedure w/o evaluating the operands
3. eval the expression produced by the body and return the result

example

```
(define-macro (twice f) `(~~list~~ begin ,f ,f))
```

~~opr~~ calls f twice

in a regular define, f would be evaluated once only

Do Jessica's method (work up)

- Create input `(_, first, second)
- input (- 2 1)
- output 1

Classes

don't forget self.

WWPD

>>False or None # (None is bad value; it is returned)
>>

range(8) = 1, 2, 3, 4, 5, 6, 7

SQL

- often used with (Count, MAX, MIN, SUM, AVG) to group result set by unique column entries
  GROUP BY ↑
  HAVING
  used with group by: filters group by

Scheme

let: local binding; stuff under can reference this binding (probably will)

append: basically (2) + (3 4) = (2 3 4)

(define x 1) x=1

(define (x) 1) x=lambda: 1

Generators

yield - can yield a pointer; thus that yielded value can be mutated! Later

if yields from a generator/iterator - use yield from

map(f, iterable)
returns iterator that maps f to iterable

filter(f, iterable)
returns iterator that iterates through value that pass f

Print/Repr

evaluates deep

x=2
print([1, x]) → [1, 2]

print(x) = print(str(x))

>> x → print(repr(x))

if str is called w/ no str-func, repr take over