[10/21/24]

RECURSION vs. ITERATION - when to use recursion

- natural recursion problem definition
- tree-like data
- list-like data

LINKED LIST vs. LIST:

- linked list tuple of tuples is IMMUTABLE!
- regular list: have to copy before copying
- list[1:] takes longer than linked_list[1]
- linked list better for recursion
- linked list is hard to index into (list[2] vs. linked_list[1][1][0])
- linked list hard to find true len (len(list) = #, len(linked_list) = 2)

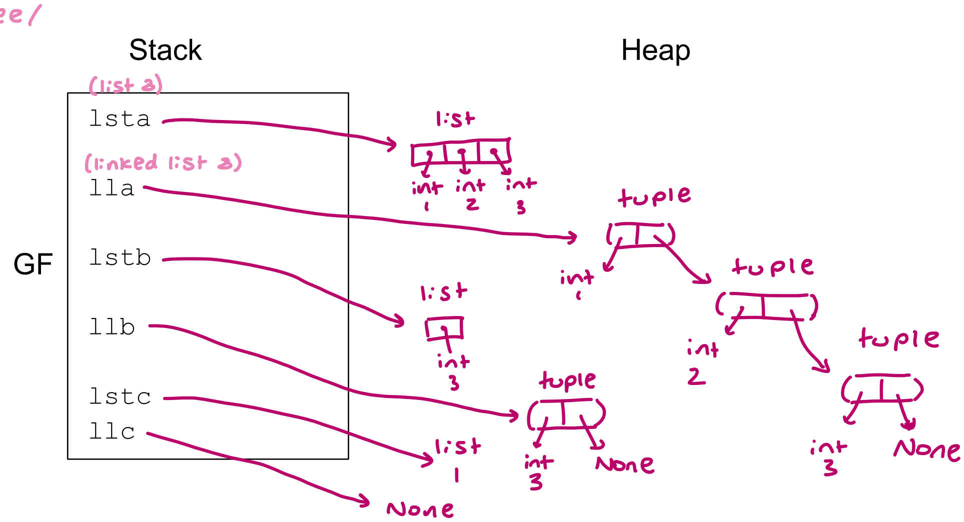
6.101 Recitation 11: Iteration and Recursion with Linked Lists **10/21/24**

Question 1: For today's recitation we will define an empty linked list as `None`, and a non-empty linked list as a length two tuple of `(element, linked_list)`.

Complete the environment diagram to represent the execution of the code below.

*linked list looks like tree/ step ladder

```
lsta = [1, 2, 3]
lla = (1, (2, (3, None)))

lstb = [3]
llb = (3, None)

lstc = []
llc = None
```



Question 2: Fill in the body of the functions below:

import doctest

```
def first(ll):
    """
    returns the first element of a non-empty linked list

    >>> first( (5, (10, (15, None))) )
    5
    """
    return ll[0]
```

doctests!!
good things to be aware of
(provides documentation AND testing)
→ there's a way to run all doctests or pick a certain one to run

```
def rest(ll):
    """
    returns the rest of a nonempty linked list
    (omitting the first element)

    >>> rest( (5, (10, (15, None))) )
    (10, (15, None))
    """
    return ll[1]
```

Why would these helper functions be useful?

- good for clarity
-

Question 3: Implement the following functions recursively, and then iteratively:

```
def ll_len(ll):
    """
    get the length of a linked list

    >>> ll_len( ('a',('b',None)) )
    2
    """
    # RECURSION:
    if ll is None:   # if not ll
        return 0

    return 1 + ll_len(ll[1])      # or: return 1+ ll_len(rest(ll))
```

base case → `if ll is None:`

recursion → `return 1 + ll_len(ll[1])`

rest: helper function we wrote

```
# ITERATIVELY:
count = 0
while ll is not None:
    ll = rest(ll)
    count += 1
return count
```

HOW DOES THIS NOT MUTATE ll??

F1: ll, count → points @ different tuples but not changing b/c not assigning; 0, 1, etc

RECURSION:

ll_len(('a',('b', None))) → 2

↳ 1 + ll_len(('b', None)) → 1

↳ 1 + ll_len(None) → 0

↳ 0

```
def ll_get(ll, i):
    """
    get the ith element of a linked list

    >>> ll_get( ('a',('b',None)), 1)
    'b'
    """
    # RECURSIVE:
    if i >= len(ll) or i<-ll_len(ll)   # if index too big OR you've added the length to negative length and it's still negative
        raise IndexError('linked list index {i} out of range')

    if i < 0:
        i += ll_len(ll)

    if i==0:
        return first(ll)

    return ll_get(rest(ll), i-1)
```

BASE CASE

*can use online to reference doctests & what is happening

```
original_i = i
length = 11_len(ll)
if i >= len(ll) or i<-11_len(ll)
  raise IndexError(f'linked list index {original_i} out of range')
return helper(rest(ll), i-1)

def helper(ll, i):
    if i==0:
      . . . .
      etc.
```

HOISTING! having recursive stuff in seperate function

[10/23/24]

WHY LINKED LIST?

- minesweeper lab has lots of nested lists - linked lists good for nested structures
- inherently recursive structure - easier to split
- list[1:] makes new copy while linked_list[1] does NOT make copy
- linked list immutable
- LISP implements linked list

6.101 Recitation 12: Iteration and Recursion with Linked Lists Part 2 10/23/24

#order matters
make_ll(n, *elements) → make_ll(1, 2, 3) → n=1, elements=(2,3)

Question 1: Implement the following function recursively, and then iteratively.

```
def make_ll(*elements):
    """
    given an arbitrary number of elements as arguments,
    make a linked-list of (first,rest) pairs

    >>> make_ll(1,2,3)
    (1, (2, (3, None)))
    """
```

makes a tuple of the elements
(unpacking) → make_ll(1,2,3) → elements = (1,2,3)
make_ll() → elements = ()

```
    #RECURSIVE
    if not elements:   ← if elements is empty list
        return None
    return (elements[0], make_ll(*elements[1:]))
```

elements (1,2,3)
make_ll ((2,3))
((2,3), _)
w/o star, will add element tuple into linked list instead of individual el.s

```
    #ITERATIVE:
    ll = None
    for elt in reversed(elements):
        ll = (elt, ll)
    return ll
```

doesn't create a copy while elements[::-1] does

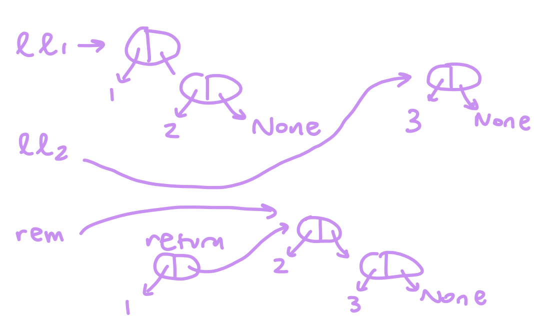
Question 2: Implement the following function recursively, and then iteratively.

```
def ll_concat(ll1, ll2):
    """
    return a new linked list that concatenates two linked lists

    >>> ll_concat(make_ll(1), make_ll(2,3))
    (1, (2, (3, None)))
    >>> ll_concat(None, make_ll(4,5))
    (4, (5, None))
    """
```

*NOTE: doctests will FAIL if you have print statements!!

```
    #RECURSIVE:
    if ll1 is None:
        return ll2
    if ll2 is None
        return ll1
    else:
        remainder = ll_concat(rest(ll1), ll2)
        return (first(ll1), remainder)
```

like adding empty list to list → [] + [4,5]




```
    #ITERATE - much messier!
    new_ll = ll2
    for i in reversed(range(ll_len(ll1))):
        new_ll = (ll_get(ll1, i), new_ll)
    return new_ll
```

Question 3: Implement the following function recursively, and then iteratively.

```
def ll_reverse(ll, so_far = None):
    """
    return a new reversed linked list

    >>> ll_reverse(make_ll(1,2,3))
    (3, (2, (1, None)))
    """
    #RECURSIVE
    if not ll:
        return None
    return ll_reverse(rest(ll), (first(ll), so_far))

    #ITERATIVELY
    new_ll = None
    while ll:                  can't do for loop
        new_ll = (elt, ll)     (or else would just be a linked list then stop)
        ll = rest(ll)
    return new_ll
```

Question 4: Implement the following function recursively, and then iteratively.

```
def ll_elements(ll):
    """
    return a generator that yields each element in a linked list
    >>> ll_gen = ll_elements(make_ll(1, 2, 3))
    >>> next(ll_gen)
    1
    >>> list(ll_gen)
    [2, 3]
    """
    while ll:
        yield first(ll)
        ll = rest(ll)
```

GENERATOR: play/pause button.
- yields then will keep playing
- NO element is retuned until next() is called
- generator is created then stays there

```
ll_gen = ll_elements(make_ll(1,2,3))
generator object (doesn't show object)

print(next(ll_gen)) ← forces ll_gen to run until yield
```

10/28/24

REFACTORING: improving code w/o changing its function

CODE SMELLS: some engineers talk ab. code having good/bad "smell" (syn. for style)

3 R'S OF REFACTORING:

→ READABILITY (documentation + style)
→ avoid REPETITION (Don't repeat yourself, make helper functions, etc.)
→ consider RUNTIME (efficiency)

REFACTOR PROCESS:

1) understand
2) make a plan
3) implement plan
4) look back

Question 1: What strategy did you use when refactoring the 2d-version of minesweeper?

```
def dig_2d(game, row, col, check-victory = True)
    # game is over, do nothing
    if game["state"] == "defeat" or game["state"] == "victory":
        game["state"] = game["state"]  # keep the state the same
        return 0
```

if game[state] != "ongoing":
    return 0

```
#checking if we dug a mine & lost
    if game["board"][row][col] == ".":      (board-val)
        game["visible"][row][col] = True    (struck)
        game["state"] = "defeat"
        return 1 revealed
```

board-val = game[board][row][col]
visible-val = game[visible][row][col]

```
#game state check - don't need to do it before
    num_revealed_mines = 0
    num_revealed_squares = 0
    for r in range(game["dimensions"][0]):
        for c in range(game["dimensions"][1]):
            if game["board"][r][c] == ".":
                if game["visible"][r][c] == True:
                    num_revealed_mines += 1
            elif game["visible"][r][c] == False:
                num_revealed_squares += 1
    if num_revealed_mines != 0:
        # if num_revealed_mines is not equal to zero, set the game state to
        # defeat and return 0
        game["state"] = "defeat"
        return 0
    if num_revealed_squares == 0:
        game["state"] = "victory"
        return 0

    if game["visible"][row][col] != True:      (visible-val)
        game["visible"][row][col] = True
        revealed = 1
    else:
        return 0

    if game["board"][row][col] == 0:      (board-val)
        nrows, ncolumns = game["dimensions"]
        if 0 <= row - 1 < nrows:
            if 0 <= col - 1 < ncolumns:
                if game["board"][row - 1][col - 1] != ".":
                    if game["visible"][row - 1][col - 1] == False:
                        revealed += dig_2d(game, row - 1, col - 1)
        # ... some code that was copy / paste / modify omitted
        if 0 <= row + 1 < nrows:
            if 0 <= col + 1 < ncolumns:
                if game["board"][row + 1][col + 1] != ".":
                    if game["visible"][row + 1][col + 1] == False:
                        revealed += dig_2d(game, row + 1, col + 1)
```

#checking for 9 neighbors surrounding the board value

already making sure there are no mines around

outside of loop already setting visible board to True

```
for r in range(max(row-1,0), min(nrows, rows+2)):
    for c in range(max(col-1,0), min(ncolumns, cols+2)):
        revealed += dig_2d(game, r, c)
```

Combining oob condition w/ getting neighbors

```
    num_revealed_mines = 0  # set number of mines to 0
    num_revealed_squares = 0
    for r in range(game["dimensions"][0]):
        # for each r,
        for c in range(game["dimensions"][1]):
            # for each c,
            if game["board"][r][c] == ".":
                if game["visible"][r][c] == True:
                    # if the game visible is True, and the board is '.',
                    # add 1 to mines revealed
                    num_revealed_mines += 1
            elif game["visible"][r][c] == False:
                num_revealed_squares += 1    ] return revealed
?? bad_squares = num_revealed_mines + num_revealed_squares
    if bad_squares > 0:      (num_revealed_squares == 0)
        game["state"] = "ongoing"
        return revealed
    else:
        game["state"] = "victory"
        return revealed
```

if game[board][r][c] != '.'
and not game[visible][r][c]:
    return revealed

← short circuiting

EFFICIENCY:

- when making smthg. a set, it takes a long time (as long as # elements in set)

  mines = set(tuple(mine) for mine in mines)
  for r, c in mines:
      ....

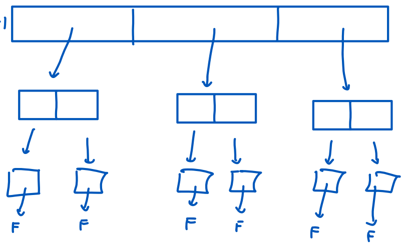
  create set outside the loop to reduce runtime

- storing values vs. function calls

Question 2: What are instances of tree-like, graph-like, and list-like recursion in the mines lab?

dimensions: 3x2x1

WATCH OUT for aliasing!



board is tree-like (each dimension is a seperate subtree)

GRAPH-LIKE:

looking for neighbors if the board value is 0

→ the "visited set" is the visible board in this case

(to prevent inf. recursion of going back / forth)

LIST-LIKE:

→ get / set value to peel off the 1st coordinate (dimension)

first : coord[0] , rest : coord[1:]

→ get neighbors

→ get all coordinates

Question 3: Below is a recursive all_coords function that returns a list of tuple coordinates. Modify the code below to make this function into an efficient generator.

```
def all_coords(dimensions):
    """
    A function that generates all possible coordinates in a given board.
    """
    if len(dimensions) == 1:
        return [(x,) for x in range(dimensions[0])]

    first = all_coords(dimensions[:1])
    rest = all_coords(dimensions[1:])
    result = []
    for start in first
        for end in rest:
            result.append(start + end)
    return result
```

10/30/24 - BACKTRACKING

1) UNDERSTAND the problem
   - what am i trying to satisfy?
      - what are my choices?
   - what are my constraints?
   - draw decision tree

2) make the plan
   - use "recipe," fill in blanks

3) implement the plan (code)

4) look back (optimize)

Question 1: You ordered food from SuperEats for you and your friends. SuperEats delivered a variety of entrees with varying quantities. Your friends have given you their unordered preferences for which entrees they like. As the host, you are trying to determine a way to assign the delivered food to your so they can all get one of their preferred dishes.

No solution example (there is not enough food for everyone): SATISFY: getting the food

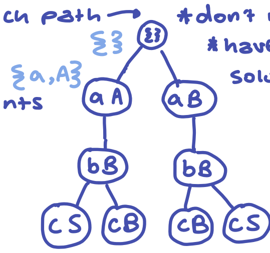
```
people = {'alex': ['Acai', 'Burger'],
          'bob': ['Burger'],
          'cam': ['Burger', 'Salad']}
food = {'Burger': 2, 'Salad': 0, 'Acai': 0}
```

Solution example (alex and bob can get burgers and cam can get a salad):

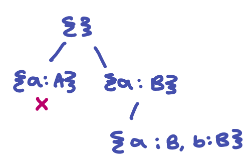
```
people = {'alex': ['Acai', 'Burger'],
          'bob': ['Burger'],
          'cam': ['Burger', 'Salad']}
food = {'Burger': 2, 'Salad': 1, 'Acai': 0}
```

Discuss with someone around you how you would approach solving this problem using different graph search methods:

- brute force search → generate all possibilities then check each
- BFS → start from a person then explore each path → *don't need visited set b/c not cycleable!
  *have to check each path even if no solution exists - not ideal
- DFS → pop(-1) instead; will be less memory use
- Backtracking → similar to DFS but w/ constraints



*recursion has DFS function



BRUTE FORCE (exhaustive)

Solution:

- explore all food items/ppl

BFS:

- remove nodes from opposite sides of agenda
- explores shortest paths 1st

DFS:

- explore current then backtrack

BACKTRACK

- similar to DFS but backtrack as soon as we realize it's not worth continuing down some branch
  - good for problems w/ constraints

Question 2: Fill in the body of the feed function below.

```
def feed(people, foods):
    """
    Given people who are hungry and the available food supplies, find a mapping
    from people to available foods they prefer if one exists.

    Parameters:
        people: a dictionary mapping a name to a list of their preferred foods
        food: a dictionary mapping available foods to their quantities

    Returns:
        Dictionary mapping person to assigned food if there is enough food to
        match everyone's preferences. None otherwise.

    >>> people = {'alex': ['oreo', 'chocolate'], 'bobbie’: ['vanilla']}
    >>> feed(people, {'oreo': 1, 'vanilla': 1}) == {'alex':'oreo','bobbie’: 'vanilla'}
    True
    >>> feed(people, {'oreo': 1, 'ketchup': 1}) == None
    True
    """
    # if nothing left to satisfy → success!!
    if not people:
        return {}

    # choose one thing to satisfy
    person = min(people, key=lambda p: len(people[p])) # take the most constrained person (min len)
    ~~person = next(iter(people))~~ # same as list(people.keys())[0]

    # look through choices to satisfy
    for food in people[person]: # food has to be in person preferences
        # non-zero food
        if foods.get(food, 0) > 0: # prevents KeyError (default 0)
            ~~people.pop(person)~~
            new_people = {p: f for p, f in people.items()}
            new_foods = {f: i for f, i in foods.items()}
            new_foods[food] -= 1  # -1 food available
            result = feed(new_people, new_foods)
            if result is not None: # success
                result[person] = food
                return result

    # if no valid choices - FAILURE
    return None
```

11/4/24

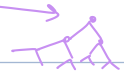
SAT Solver overview:

→ satisfying_assignment w/ backtracking recipe

→ optimizing
- prune tree
- avoid cycles
- avoid extra copying

f = (a or b or c) and (b or not a) and (not b) ← CNF

satisfying_assignment (SA)



etc.

RULES: * trying to satisfy the CLAUSE!
- no clauses left → solved!
- empty clause = unsolvable
- variable can only have one value

* computers NOT fast enough to keep up w/ 2^n

(try to minimize tree branches)

SA (f)

var, val = 'a', True

a^T ↙ $f_1 = [[b^T], [b^F]]$

b^T → $f_2 = [[\,]]$

c^T → $f_3 = [[b^T, a^F] [b^F]]$

SA (f_1)

b^T ↙ $F_2 = [[\,]]$ X backtrack

SA (f_3)

b^T ↙ [[ ]] X

a^F ↘ $[[b^T]]$ etc.

etc

nondeterministic polynomial time — polynomial time

NP = P?

$n^k <<< 2^n$

much more efficient

* INEFFICIENT b/c unit clause B^F MUST work!
- loop through unit clauses or the T/F of shortest clauses

a^T ↙ • ← similar to binary tree

$f_1 [[b^T] [b^F]]$

SA (f_1)

b^T ↙ [[ ]] b^F ↘ [[ ]]

$f_3 = [[b^T, c^T], [b^F]]$

b^T ↙ [[ ]]

b^F ↘ $[[c^T]]$

SA (f_4)

c^T ↙ [ ] ✓

YIELD vs. YIELD FROM:

- yield from is basically a for loop + yield

```
def my_gen():
    for el in [1,2,3]
        yield el
```
} gets 1, 2, 3

```
    yield from [1,2,3]
```
} → also gets 1, 2, 3

- yield returns a single val from generator function
- yield from yields all values from iterator/gen.

6.101 Recitation 15: SAT Solver Wrap-up **11/4/24**

Question 1: Applying Polya's problem solving method to satisfying_assignment:

format: ((p, f), True)

instead of combine (ppl name kids weird things)

1] Understand the problem

- What are the rules / constraints of satisfying_assignment?
- What are we trying to satisfy?
- What kind of decision tree do we end up with?

similar to boolify:
- food mapped to quantity similar to room to capacity
- name & preferred food

RULES:
1: each person only needs 1 food
2: can only give out x number of food
3: ppl need to eat exactly 1 food

2] Make a plan

- How do we apply our general recipe for backtracking to satisfying_assignment?
- Will we need any helper functions to be useful?
  - → preferred_foods() (rule 1)
  - → 1 function for rule 2 person_one_food(), rule 3 at_most()

with format ((person, food), bool), how to take result of satisfying_assignment and output final result?

```
if result is None:
    return None
else:
    assign = {}
    for (people, food), value in result.items():
        if value:
            assign[people] = food
    return assign
```

3] Implement the plan

4] Look back

- How can we optimize this search?

Question 2: What is the result of calling satisfying assignment on the formula below? How does using the unit clause optimization make solving this formula faster?

```
formula  = [
            [('a', True), ('b', True), ('c', True)],
            [('b', True), ('a', False)],
            [('b', False),],
           ]
result = satisfying_assignment(formula)
```

Question 3: How can we solve our potluck from last Wednesday using satisfying_assignment?

Question 4: implement all_combinations as a generator function

```
def all_combinations(elements, N):
    """
    Given a list of hashable elements (with no duplicates) and N
    (the size of each combination), make a generator that outputs
    length-N tuples of all combinations of the elements

    >>> sorted(all_combinations([1, 2, 3, 4], 0)) == [()]
    True
    >>> sorted(all_combinations([1, 2, 3, 4], 1)) == [(1,), (2,), (3,), (4,)]
    True
    >>> y = all_combinations([1, 2, 3, 4], 2)
    >>> sorted(y) == [(1, 2), (1, 3), (1, 4), (2, 3), (2, 4), (3, 4)]
    True
    >>> z = all_combinations([1, 2, 3, 4], 3)
    >>> sorted(z) == [(1, 2, 3), (1, 2, 4), (1, 3, 4), (2, 3, 4)]
    True
    >>> sorted(all_combinations([1, 2, 3, 4], 4)) == [(1, 2, 3, 4)]
    True
    >>> sorted(all_combinations([1, 2, 3, 4], 5)) == []
    True
    """
```

WITH generators: think about what you want to append!!

*if generators are scary, return list first then do a generator that yields from

GENERATOR:

```
if N == 0:
    yield [()]
elif not elements:
    yield []
else:
    first = elements[0]
    rest = elements[1:]

    yield from all_combinations(rest, N)

    for combo in all_combinations(rest, N-1)
        yield (first,) + combo
```

NOT GENERATOR:

```
if N == 0:
    return [()]
elif not elements:
    return []
else:
    first = elements[0]
    rest = elements[1:]

    all_combos = all_combinations(rest, N)

    for combo in all_combinations(rest, N-1):
        all_combos.append((first,) + combo)
    return all_combos
```

WHEN TO USE self.__ ?

→ when you want to keep a variable

when print(obj) in a class, calls __str__

__repr__ also

6.101 Recitation 16: Classes & N-Body Physics Simulation 11/6/24

abstraction, planet simulation

Question 1: Today we're going to build a planet physics simulation (see video.) Discuss with a neighbor: what classes would be a good idea to implement? What attributes and methods should each class have?

$\vec{v}_i[t+1] \approx \vec{v}_i[t] + \vec{a}_i[t]\,dt$

$\vec{x}_i[t+1] \approx \vec{x}_i[t] + \vec{v}_i[t]\,dt$

$\vec{a}_i = \frac{\sum_j \vec{F}_{ij}}{m_i} \rightarrow \vec{F}_{ij} = \frac{G \cdot m_i \cdot m_j}{\|r\|^2}\hat{r}$

CLASSES:	ATTRIBUTES:	METHOD:
Planet	mass position (vector) ← 2D space velocity	· forces - parameters: planet, list of other planets applying $\vec{F}$
Simulation	list of planets	apply timestep dt
→ vector	nd coord list	+, -, /, *

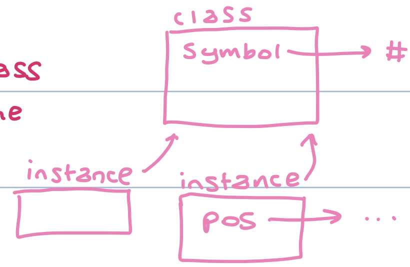
will implement these today

Question 2: Fill in the missing code below for the following Vector class methods:

```
class Vector:          first arg. always
                       the instanece of self

    def __init__(self, coords):
        # each Vector object has a nd tuple of coords
        self.coords = tuple(coords)

    def __repr__(self):
        # repr(Vector([0, -4]))  -> 'Vector((0, -4))'
        return f'{self.__class__.__name__}({self.coords})'

                                      can use + instead
    def add(self, other):  +          of .add b/c DUNDER!!
        # Vector([1, 2]).add(Vector([1, 0]))  -> Vector((2, 2))
        return Vector([i+j for i,j in zip(self.coords, other.coords)])

    def sub(self, other):
        # Vector([1, 2]).sub(Vector([1, 0]))  -> Vector((0, 2))
        return Vector([i-j for i,j in zip(self.coords, other.coords)])

                        num
    def scale(self, other):
        # Vector([1, 2]).scale(5)   -> Vector((5, 10))
        return Vector([i*num for i,j in self.coords])

                      num
    def div(self, other):  .truediv
        # Vector((4, 2)).div(2)   -> Vector((2.0, 1.0))
        return Vector([i/num for i,j in self.coords])
        OR
        return self.scale(1/num)

    def magnitude(self):   abs(vec....)
        # Vector((3, 4)).magnitude() -> 5.0
        return sum([i**2 for i in self.coords])**0.5

    def normalize(self):  # creates a unit vector in the same direction
        # Vector([3, 4]).normalize()   -> Vector([.6, .8])
        return self.div(self.magnitude())
```

always ask:
what is input/output type?

CLASS ATTRIBUTE → stored inside class
(each instance of class will share the class attribute)



Question 1: Eat That Sock is a 2D game where a human player competes with a bot to see who can score the most points in twenty seconds. Players score points by collecting socks. Socks can be different colors, with each color corresponding to a different point value. Socks randomly appear in the game, and also randomly disappear if the players don't collect them fast enough. Walls keep the players and socks in bounds.

Discuss with a neighbor: What are the different objects that we would need to represent this game? What attributes and methods would each object need?

For example Game:
- Stores all other objects
- Keeps track of time remaining
- Each time step:
  - Moves players
  - Makes socks appear / disappear
  - Renders updated game board
- Decides winner at the end

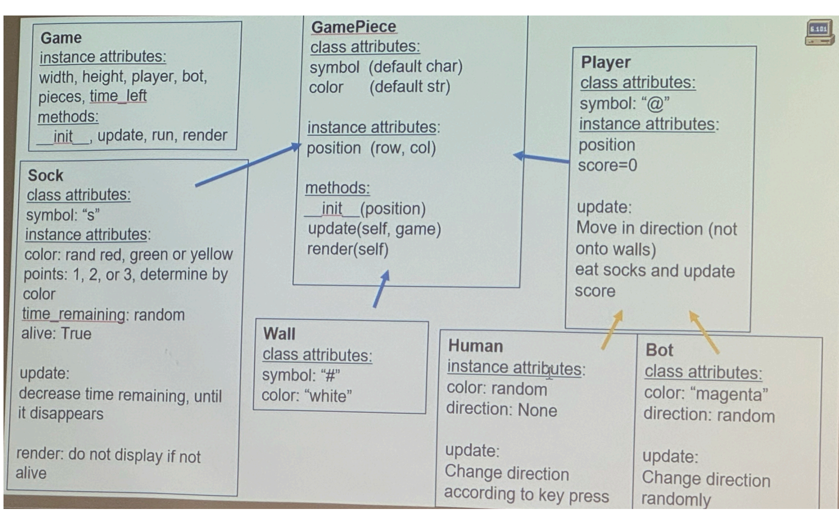
*think of classes & subclasses like a tree - parent = root node & child is a lower node.

Sock:
- position (row, col)
- color: red, yellow, green
- symbol: s
- points: red 1, green 2, yellow 3
- TTL (time until dissapear)

Player:
- position
- color (random)
- symbol: @
- render
- score
- change dir. w/ key
- move according to current dir.
- eats socks

Bot:
- position
- color (magenta)
- symbol: @
- render
- score
- move according to dir.
- randomly change dir.
- eats socks

Wall:
- position
- color (white)
- Symbol: #
- render
- doesn't move
- can't intersect w/ other objects



GAMEPIECE
Class instances:
color, symbol, alive

Question 4: Look at the blank Bot class shown below. Fill in the missing body. If implementing a method would be unnecessary, cross it out instead.

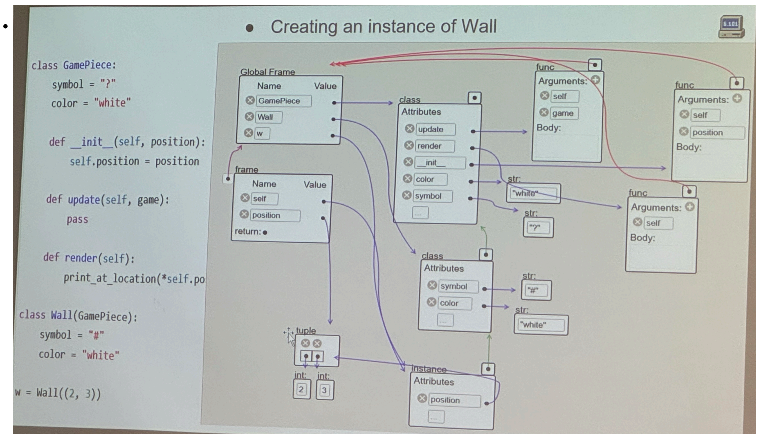
```
class Bot(Player):
    """
    A basic bot that randomly moves around the screen.
    """
    color = "magenta"


    def __init__(self, position):




    def update(self, game):
        """
        Given the recent keys pressed and the current game, update the object.
        """
        directions = [Game.UP, Game.DOWN, Game.LEFT, Game.RIGHT]
        self.direction = random.choice(directions)
        Player.update(self, game)

    def render(self):
        """
        Takes the state of the object at the end of a timestep and displays
        it to the screen.
        """
```

R17 Participation Credit **Kerberos : _______________@mit.edu**

Hand this sheet in at the end of recitation to get participation credit for today.

Question 2: Look at the blank Wall class shown below. Fill in the missing body. If implementing a method would be unnecessary, cross it out instead.

```
class Wall(GamePiece):
    """
    Static game piece represented by a white "#" symbol, which prevents
    pieces from going out of bounds.
    """
    symbol = "#"
    color = 'white"

    def __init__(self, position):



    def update(self, game):
        """
        Given the current state of the game, update the object.
        """


    def render(self):
        """
        Takes the state of the object at the end of a timestep and displays
        it to the screen.
        """
```



don't need to implement any of these functions!! all inherited from GamePiece

in render from class Gamepiece:
print_at_location(*self.position, self.symbol, self.color)

Question 3: Look at the blank Sock class shown below. Fill in the missing body. If implementing a method would be unnecessary, cross it out instead.

```
class Sock(GamePiece):
    """
    Static game piece represented by "s" that is visible for ttl timesteps
    before disappearing. Socks come in different colors, and each color is
    worth a different number of points. Players earn points if they intersect a
    sock before it disappears.

    Created by the game at a random position, initialized with a random color,
    disappears randomly.
    """
    symbol = "s"
    color_options = {"red": 1, "green": 2, "blue": 3}
    def __init__(self, position):
     self.color = random.choice(list(self.color_options.keys()))
      # super().__init__(position) works here, but NOT encouraged
      GamePiece.__init__(self, position)
      self.ttl = random.randint(10,20)

    def update(self, game):
        """
        Given the current state of the game, update the object.
        """
        self.ttl -= 1
        if self.ttl <= 0:
            self.alive = False


    def render(self): <-same code no matter what object
        """
        Takes the state of the object at the end of a timestep and displays
        it to the screen.
        """
```

11/18/24 - Symbolic algebra lab & classes/inheritance

GENERAL PURPOSE TOOLBOX: classes, functions, data structures

"NICHE" TOOLS: backtracking

WHAT IS TYPE-CHECKING:

Ex: every class has a precedence attribute (general class attribute)
- evaluate __str__ w/ precedence instead of type-checking
- don't want to have to have many classes & many if statements

Ex: Var('x') + 5 → __add__

5 + Var('x') → __radd__

IN EACH BINOP SUBCLASS... (add, sub, deriv, etc)
- simplify method
- eval method

MAKE-EXP: symbolic algebra vs. LISP

symbolic algebra:	LISP:
(x + 5)	(+ x 5)
(x + 5 + y)	(+ x 5 y)
	comments (+ x 5) ; ignore rest of line

TOKENIZE:

```
def tokenize(text):
    """
    tokens = []
    number_in_progress = ''
    for char in text:
        if '0' <= char <= '9':  ← ascii (compares if char is within these str ranges)
            number_in_progress += char
            continue  ← keep looking for number in next ind
        if number_in_progress:
            tokens.append(number_in_progress)
            number_in_progress = ''
        if 'a' <= char <= 'z': # single chars
            tokens.append(char)
        elif char in '()+-*/': # separator
            tokens.append(char)
        elif char == ' ':
            continue  ← ignore spaces
        else:
            raise Exception(f'unexpected character {char}')
```

6.101 Recitation 18: Symbolic Algebra Wrap-up 11/18/24

Question 1: Discuss with a neighbor: What are the pros and cons of using classes (as opposed to implementing the same kind of behavior using builtin datatypes and functions)? What are some real-world applications of classes?

CLASSES: vs. FUNCTIONS / BUILTINS:

+ inheritance
  +having implicit behavior
  $V_1 + V_2$ __attr__
  overwriting behavior
+ good way of organizing code

— understand

— more work/code to overwrite implicit behaviors {}

Question 2: Discuss with a neighbor: What are some real-world applications of inheritance?

- many shared behaviors / attributes b/t things (sound, pieces, animals, etc)
  - Ex: instead of implementing Sounds as dicts... class Sound, MonoSound, StereoSound

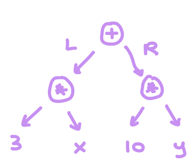
Today we're going to think about how to implement a variation of `make_expression` that has:
- nonnegative integers, like 3 or 10
- single-letter lowercase variables, like x or y or z
- fully-parenthesized binary operators, like (3*x) for +, -, /, and *
- no extra parens or missing parens
- ignores spaces

Question 4:
`make_expression(" ((3 * x)+(10*y) ) ")` =>Add (Mul (Num(3), Var('x')), Mul (Num(10), Var('y')))

→ tokenized (split into chars in list)

→ parsed

* TREE-LIKE RECURSION!
- left, right branches off
- Var, Num are leaf nodes



Question 5: Discuss with a neighbor: How would we go about `tokenize`?

→ looking for seperator or num

→ example code above

Question 6: Discuss with a neighbor: How would we go about `parse`? What recursive pattern are we using (list-like, tree-like, graph-like)?

```
def parse(tokens):  # tokens is a list of strings

    def parse-exp(index):
        # base cases = leaves of tree
        # if number
        if "0" <= tokens[index][0] <= "9"
            return Num(int(tokens[index])), index + 1
        # if b/t a-z (var)
        if "a" <= tokens[index][0] <= "z"
            return Var(int(tokens[index])), index + 1
        # left & right sides
        left - exp, index_2 = parse _exp (index + 1)
        operator = tokens [index_2]  # + - * /
        right- exp, index - 3 = parse - exp(index - 2 + 1)

        # find operators & assign
        op_lookup = {'+': Add, '-': Sub, '/': Div, '*': Mul}
        operator_class = op_lookup[operator]
        return operator_class(left-exp, right_exp), index3 + 1

    return parse_exp(0)
```

Bonus exercises:

- Modify `make_expression` to allow for repeated operations in the same parens, i.e., `make_expression('(3 + 4 + x + (y - 5 - 2 - z) + (3 * v))')`
- Generators: Modify tokenize to output a generator and modify parse to take in a generator.
- Modify `make_expression` to detect and raise a custom error when the expression is malformed, i.e., `make_expression('(3 + 4')`

11/20/24:

WHY USE INTERPRETERS (LISP):
- help understand languages you already know
- idea: interpreter is just another program

INTERPRETER → program converting high-level language to machine code & then executes it on the go
- ex: Scheme

COMPILER → "translator" – program converting a program in one language to another

PYTHON VS. SCHEME (interpreter)

```
# python
def mag (x,y):
    return sqrt(x*x + y*y)
magnitude (3,4)
```

```
; scheme
(define (mag x y)
   (sqrt (+ (* x x)
            (* y y) ) )
)

(magnitude 3 4)
```

NEXT LAB → conditional statements
- recursion
- can use for any coding language

~LAB~

PROGRAM → Tokenize → Parse → Evaluate → OUTPUT

Ex: '(-(+3 2) 5); test comment'

tokenize: ['(', '-', '(', '+', '3', '2', ')', '5', ')']

parse: ['-', ['+', 3, 2], 5]

evaluate: 0

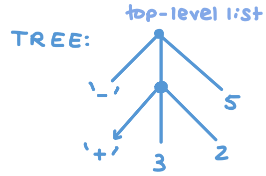
Question 1: Discuss with someone near you: what are the similarities and differences between tokenizing and parsing in the Symbolic Algebra lab and the LISP lab?

LISP:
- deals w/ comments ;
- can define a long, multi-letter variable w/ assignment
- evaluate different lines
  - parse/tokenize entire file

TREE-LIKE LISP
- leaf nodes (base exp): numbers/variables
- S-expressions: EX: '(-(+3 2) 5); test comment'

top-level list

TREE:



EVALUATE:

[sum, 3, 2] → 5

[sub, 5, 5] → 0

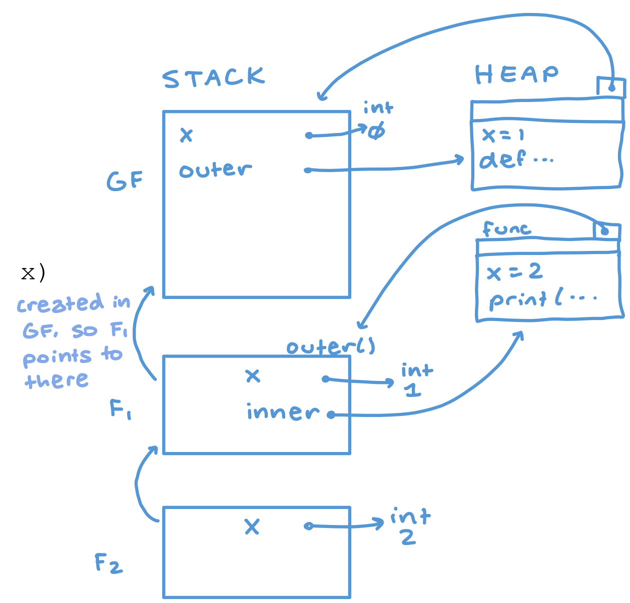
- EVALUATE:
  - check type of the char & go from there
  - str, num, list

  [sum, 3, 2] → 5

  [sub, 5, 5] → 0 ✓

Question 2: What will the code below output? Draw an environment diagram to represent the program execution.

```
x = 0
def outer():
    x = 1
    def inner():
        x = 2
        print("inner:", x)

    inner()
    print("outer:", x)

outer()
print("global:", x)
```



inner: 2
outer: 1
global: 0

b) what if x=2 is commented out?

inner: 1
outer: 1
global: 0

← look into parent frame

c) if inner() function is moved out? (& x=2 still commented out?)

inner: 0
outer: 1
global: 0

☆ Relatedly, what properties and methods would a Python class representing a frame object need? What about a function object?

FRAME CLASS

attribute:
- parent
- variables/values

methods:
- lookup (input: var return: value)

FUNCTION CLASS

attribute:
- enclosing frame
- parameters
- body

methods:
- call (input: arguments) ← python has custom __call__ method

* can check if something is callable(...) (if has the call method)

Question 3: Rewrite each of the Python expressions below in Scheme.

```
# example 1
(5 + 4) / (7 - 3 - 2 - 1) / 2
```

(/ (/ (+ 5 4) (- 7 3 2 1)) 2)

```
# example 2
(lambda x: x*x)(4)
```

← function call surrounds

((lambda (x) (* x x)) 4)

```
# example 3
def area(r):
     return 3.14 * r ** 2
x = area(5)
y = x
```

(define area (lambda (r) (* 3.14 r r)))

(define x (area (5)))

(define y x) ← var name is str, exp. has ( ) around it

```
# example 4
def four():
    return 2 + 2
four()
```

??

(define four () (+ 2 2))

four

11/25/24

LISP part 1 - diff. b/t function call & special form

WHY SCHEME?

- good practice to see similar/diff. b/t languages
- no looping in scheme! recursion only.
- useful for understanding test cases

6.101 Recitation 20: Lisp part 1 Wrap-Up 11/25/24

Question 1: For each of the four statements written in Python below:
-What is the equivalent expression written in Scheme?
-What will the output of interpreting that expression be?
-How many times will evaluate be called in the course of interpreting that expression?
Note, example A has been completed for you.

(lambda (x) (* x x))
→ calls eval. once & stores the body
(+ 3 4 x)
→ eval. 5 times (looks @ each element)

```
; Example A:
; x = 4            ; provided Python code
(define x 4)       ; Scheme equivalent
; output: 4
; # calls to evaluate: 2 – why?
```
→ evaluates x & evaluates 4

```
; Example B:
; y = x - 1
```

```
; output:
```
(define y (- x 1))
```
; # calls to evaluate:
```
3 (eval whole thing, then y & (- x 1))

```
; Example C:
; square = lambda s: s * s
```

```
; output:
```
(define s (lambda (s) (* s s)))
```
; # calls to evaluate:
```
2 (for define & lambda)

(define (square x) (* x x))
calls: 1 (just sets square to body)

```
; Example D:
; z = square(x) + square(y)
```

```
; output:
```
(define z (+ (square x) (square y)))
```
; # calls to evaluate:
```
17 (square x creates new frame & calls eval. many times for the body, etc.)
(evaluator does a lot under the hood)

Question 2: The following Scheme code comes from test_inputs/21.scm. Convert this scheme program into an equivalent Python program.

function call (parentheses)

```
(define (call x) (x)) function object
(call (lambda () 2))  2 ←creating func. & immediately calling
(define (spam) (call (lambda () 2))) function object
(call spam) 2
(call call) SchemeEvaluationError (bad args)
(call) SchemeEvaluationError (missing arg)
```

spam = lambda funct.
spam()
call (spam) = 2

Question 3: Syntax errors occur when code breaks the rules the rules define the combinations of symbols that are considered to be correctly structured expressions in a programming language. Syntax errors are generally caught during parsing and prevent any lines of code from being interpreted / evaluated.

Discuss with a neighbor: what are common kinds of syntax errors in Python?

- indentation
- missing a colon

Thinking back to our Symbolic Algebra recitation last Monday, what are some examples of things that would cause syntax errors?

- missing round brackets
  - →can count parentheses, but don't have to.

As a reminder, our variation of `make_expression` last Monday handled expressions with:
- nonnegative integers, like 3 or 10
- single-letter lowercase variables, like x or y or z
- fully-parenthesized binary operators, like (3*x) for +, -, /, and *
- no extra parens or missing parens
- ignores spaces

SCHEME
expr

- atomic expression
  - var (str)
  - # (int, float)
- s-expression
  - base exp
    - (func args...)
    - evaluate everything
    - built-ins
    - + - / * < >
    - <= >= not #t
    - #f equal?
  - special form
    - (define var expr)
    - (lambda (param.) expr)
    - if
    - and
    - or

6.101 Recitation 21: Scheme conditionals and recursion **12/2/24**

Question 1: Rewrite the `abs` function using a lambda expression in Python.

```
def abs(x):
    if x < 0:
        return -x
    else:
        return x
```

```
abs = lambda x: -x if x<0 else x
```

Question 1a: Now write the `abs` function in Scheme.

```
(define abs (lambda (x)
    (if (< x 0)
        (* -1 x)
              x))
```

Question 2: Why do `if`, `and`, and `or` need to be special forms in Scheme?

- important that both conditionals (T/F) don't get evaluated
    - if: only one of two options should happen
- and & or → short circuit (stops early)
    - and: stops as soon as F
    - or: stops as soon as T

Question 3: Write the `sign` function below in Scheme.

```
def sign(x):
    if x < 0:
        return -1
    elif x > 0:
        return 1
    else:
        return 0
```

```
(define sign x
    (if (< 0 x) -1
        (if (> 0 x) 1
                    0))
```

6.101 Recitation 21: Scheme conditionals and recursion **12/2/24**

Question 4: Write `sum_squares` below in Python without using loops. This function should return the same input as the original version for all integers n <= the recursion limit.

```
def sum_squares(n):
        total = 0
        for i in range(n+1):
            total += i*i
        return total

def sum_squares(n): # no loops!

    if n == 0:
        return 0
    return n*n + sum_squares(n-1)
```

Question 4a: Write `sum_squares` below in Scheme:

```
(define sum_squares n)
    (if (equal? n 0)
        0
        (+ (sum_squares (- n 1)) (* n n)))
    )
)
```

* begin statement

Question 3a: Write the `sqrt` function below in Python without using loops. This function should return the same input as the original version for all integers n <= the recursion limit.

```
def sqrt(x, epsilon):
    guess = x / 2
    while abs(guess ** 2 - x) > epsilon:
        guess = (guess + x/guess) / 2
    return guess

def sqrt(x, epsilon):
   # your code below
   guess = x/2
   if abs(guess ** 2 - x) > epsilon:
       guess = (guess + x/guess)/2
       return sqrt(guess, epsilon)
   return guess
```

Question 3b: Write the `sqrt` function in Scheme.

12/4/24

- EXPR
  - atomic
    - var
    - # (int, float)
  - s-expr.
    - basics expr (eval.) (*, -, etc)
    - special form (define, lambda, if, and, or)

PYTHON → SCHEME
- remove loops
- replace vars
- if : __ if __ else __

6.101 Recitation 22: Linked Lists in Python and Scheme **12/4/24**

Today we'll be working with lists in Scheme, which will give us some practice with recursion and linked lists.

The table below shows the built-in list behavior that you will implement during Lisp part 2 and how it relates to Python list and linked tuple behavior that we have seen in recitation previously:

Scheme list	input	return type	Python list	linked tuple
`(list? x)`	any	boolean	`isinstance(l, list)`	
`(list ...)`	sequence<any>	linklist	`[...]`	`make_ll(...)`
`(cons x y)`	any	cons cell	`[x, y]`	`(x, y)`
`(car x)`	cons cell	any	`x[0]`	`first(x)`
`(cdr x)`	cons cell	any	`x[1:], or x[1]`	`rest(x)`
`(append ...)`	sequence<linklist>	linklist	`sum(..., [])`	
`(length x)`	linklist	int	`len(x)`	`ll_len(x)`
`(list-ref x i)`	linklist num	any	`x[i]`	`ll_get(x, i)`

Question 1: Write the `sum_list` function below in Scheme.

```
def sum_list(x):  # x is a flat list of numbers
    out = 0
    for i in x:
        out += i
    return out
```

RECURSIVE:

return (0 if not x else x[0] + sum_list(x[1:]))

```
(define (sum_list x)
    (if not x
     0
     (+ (car x) (sum_list (cdr x))
```

Question 2: Write the `sum_nested` function below in Scheme.

```
def sum_nested(x):  # x is a nested list of numbers
    out = 0
    for elt in x:
        out += sum_nested(elt) if isinstance(elt, list) else elt
    return out
```

```
(def (sum_nested x)
    (if (equal? (length x)
    0
```

Question 3: Write the `subtract_elts` function below in Scheme.

```
def subtract_elts(x1, x2):
    # x1 and x2 are flat lists of numbers that have the same length
    result = []
    for i in range(len(x1)):
      result.append(x1[i]-x2[i])
    return result
```

RECURSION:

```
def subtract_elts(x1, x2):

    def loop(i):
        if i == len(x1)
            return []
        else:
            return [x1[i] - x2[i]] + loop(i+1)]
```

SCHEME:

```
(define (subtract_elts x1 x2)
  (
    (define (loop i)
        (if (equal? i (length x1))
            ()
            (list
              (- (list-ref x1 i)
                 (list-ref x2 i))
              (loop (+ i 1))
            )
        )
    )
    0
  )
)
```

Question 4: Write the `find_max` function below in Scheme.

```
def find_max(x):
    # x is initially a non-empty nested list of numbers
    if isinstance(x, list):
        best = find_max(x[0])
        for elt in x:
            best = max(best, find_max(elt))
        return best
    return x
```

12/9/24 - last rec!

WE LEARNED:
- data structures (dict, sets, lists, iterables, generators)
- functional programming
- graph search
- recursion
- classes/inheritance
- languages/interpreters

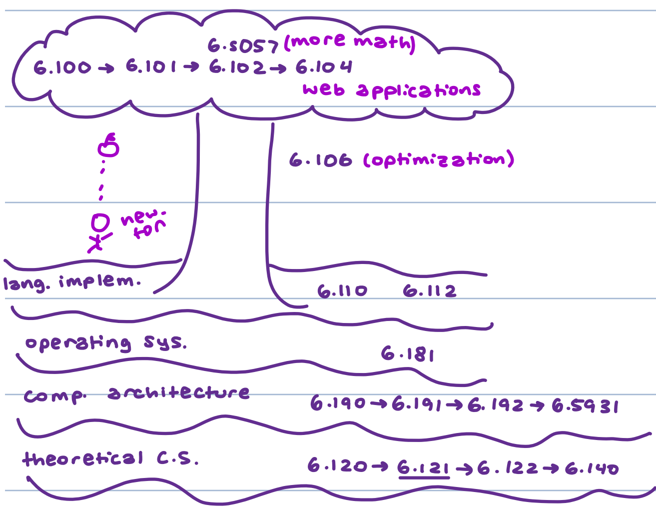
COMMON EFFICIENCY BUGS:
- superfluous computation
- suboptimal data structure design
- suboptimal algorithm design (SAD)

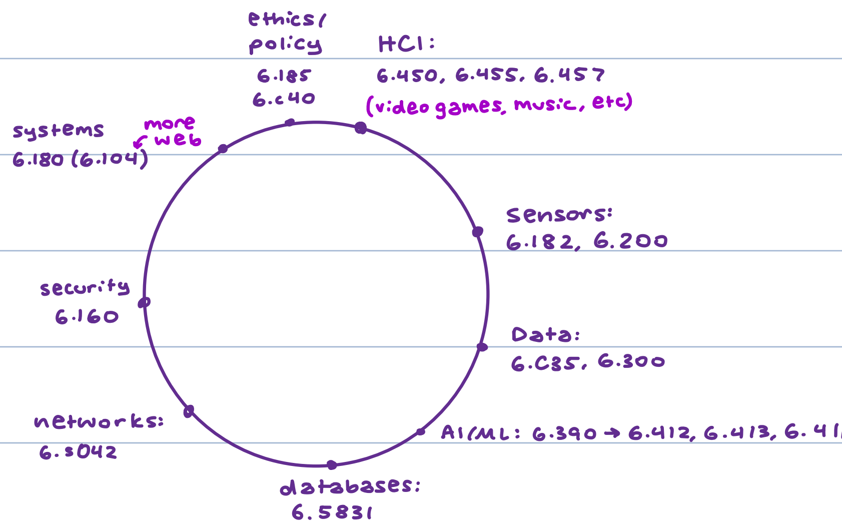
x = list(x)

$\underbrace{\text{setx = set(x)}}$

conversions b/t types are LINEAR

LOOKING FORWARD:
- use imports
- python standard lib & external packages

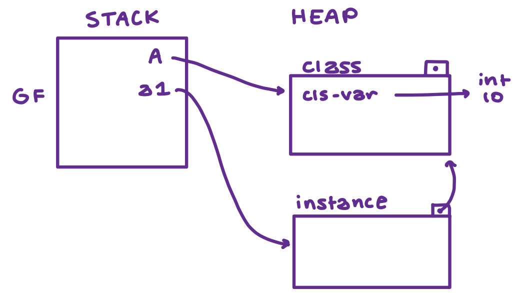




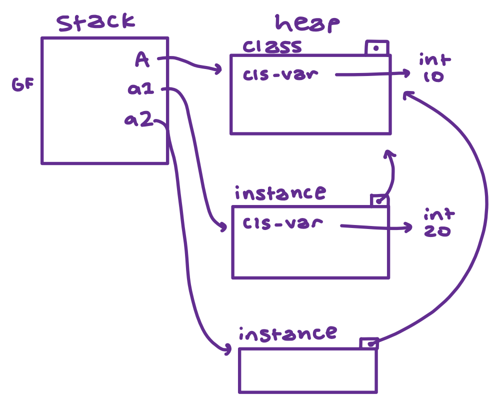
Question 1: Classes + Environment Diagrams

For each example program below, write what the output will be and draw the associated environment diagram. If the running the program would result in an error, write error instead and describe the problem.

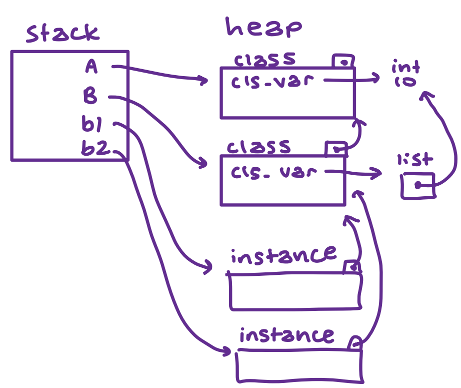
```
# example A
class A:
    cls_var = 10
a1 = A()
print(a1.cls_var) # =>  10
```



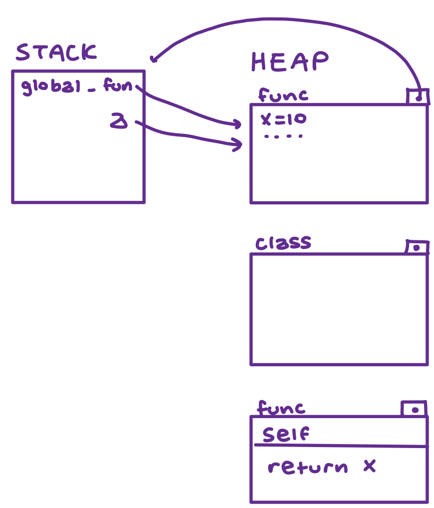

```
# example B
class A:
    cls_var = 10
a1 = A()
a2 = A()
a1.cls_var = 20
print(a1.cls_var) # =>  20

print(a2.cls_var) # =>  10
```




```
# example C
class A:
    cls_var = 10
class B(A):
    cls_var = []

b1 = B()
b2 = B()
b1.cls_var.append(A.cls_var)
print(b1.cls_var) # =>  [10]

print(b2.cls_var) # =>  [10]
```




```
# example D
def global_fun():
    x = 10
    class A:
        def a_fun(self):
            return x
    x = 20
    return A()

a = global_fun()
print(a.a_fun()) # => ________
```



Question 2: Recursion + Iteration + Generators

For each example implementation of the map function below, what would the output of the following program be? If this would result in an error, write error instead.

```
def add_to_all(map_func, k, inp_list):
    return map_func(lambda n: n + k, inp_list)

print(add_to_all(map, 3, [1, 2, 3, 4, 5]))
print(list(add_to_all(map, 3, [1, 2, 3, 4, 5])))

# example A
def map(f, inp):
    return list(f(x) for x in inp)


# example B
def map(f, inp):
    yield f(inp[0])
    yield from map(f, inp[1:])



# example C
def map(f, inp):
    if not inp:
        yield []
        return
    yield f(inp[0])
    yield from map(f, inp[1:])



# example D
def map(f, inp):
    if not inp:
        return
    yield from map(f, inp[1:])
    yield f(inp[0])



# example E
def map(f, inp):
    if not inp:
        return
    yield f(inp[0])
    yield from map(f, inp[1:])
```

Question 3: Backtracking with Tent Packing – see readme

What are the success base case(s)?

What are the failure base case(s)?

What are the recursive case(s)?

Write a description of a high-level algorithm you could use to solve the problem

Question 4: More practice

```
# example a Write the body of an infinite generator that will produce the desired output below
def fibonacci_generator(a=0, b=1):





for i, fib in zip(range(9), fibonacci_generator()):
   print(f"fib({i})={fib}")
# fib(0)=0 fib(1)=1 fib(2)=1 fib(3)=2 fib(4)=3 fib(5)=5 fib(6)=8 #fib(7)=13 fib(8)=21
```

```
# example b  -- turn this generator into a regular function
def flatten(lst):
   for item in lst:
       if isinstance(item, list):
           yield from flatten(item)
       else:
           yield item
nested_list = [1, [2, [3, 4], 5], 6, [7, 8]]
print(list(flatten(nested_list))) # [1, 2, 3, 4, 5, 6, 7, 8]
```

Question 5: BFS vs DFS + recursion

What will the program below output?

```
def dfs(graph, start):
   visited = set()

   def dfs_recursive(vertex):
       visited.add(vertex)
       yield vertex
       for neighbor in graph[vertex]:
           if neighbor not in visited:
               yield from dfs_recursive(neighbor)

   yield from dfs_recursive(start)

# Define a graph as an adjacency list
graph = {
   'A': ['B', 'C'],
   'B': ['D', 'E'],
   'C': ['F'],
   'D': [],
   'E': ['F'],
   'F': []
}

# Perform DFS traversal starting from 'A'
for vertex in dfs(graph, 'A'):
   print(vertex)



# For an extra challenge, write a recursive BFS version below. What will the #
printed output be now?
```

Summary of Readings Since Exam 1

Recursion

- base case, recursive case, combination
- recursive cases are smaller than original
- helper functions

Recursion and Iteration

- some problems naturally iterative, some recursive
- recursive patterns
  - list-like: first/rest
  - tree-like: children
  - graph-like: neighbors
- recursive helper to accumulate partial results
- recursion for DFS
- generators

Recursion with Backtracking

- a generalized graph-like search with constraints
- typical structure
  - success base case
  - failure base case
  - recursive case that reversibly tries possibilities

Custom Types

- the power of abstraction
- Python's class keyword and the environment model
- class vs instance attributes / variables
- two scoping paths: variable lookup through frames, attribute lookup through dot notation / classes

Inheritance

- subclasses and instances
- inheriting vs overlaying methods, leveraging polymorphism
- lifting shared behaviors to superclasses

Functional Programming

- converting imperative-style loops to functional-style recursion
- converting classes to nested functions
- memoization