6.1010 CHEAT SHEET- CATHERINE TU

CAREFUL:
- mutating directly on parameter

- environment diagram frames made AFTER being called

- what is the code actually doing??

- matching brackets, indents, etc.

[SETS:] - mutable

- CANT include unhashable elements (ex: NO list) (NO dict)
- O(1) search time
- auto sorts
- use hash table (hash function)

[DFS]

- add/remove elements from same side of agenda
- search one path, than another
- LIFO (similar to stack)
- guaranteed to find path, but not necessarily optimal
- may run forever on infinite graph, even if solution exists

[BFS]

- add/remove elements from opposite sides of agenda
- FIFO
- search in layers
- guarenteed to return shortest path if it exists
- can run foreve if being applied to infinite graph w/ no solution

PATH-FINDING CODE: DFS/BFS EXAMPLE

```
def find-path(neighbors-func, start-state, goal-test, bfs=True):
            ↑ possible neighbors
    """
    ...
    """
    if goal-test(start-state):
        return (start-state,)
    agenda = [(start-state,)]
    visited = {start-state}
    while agenda:
        this-path = agenda.pop(0 if bfs else -1)
        terminal-state = this-path[-1] ← extracts last state (path
        for neighbor-state in neighbors-func(terminal state):
            if neighbor-state not in visited:
                new-path = this-path + (neighbor-state,)   ← adding unvisited states to path
                if goal-test(neighbor-state):
                    return new-path
                agenda.append(new-path)
                visited.add(neighbor-state)
```

[5 DISTINCT THINGS:] GRAPHS (DFS/BFS)

- graph itself
- start state/goal condition
- candidate path (neighbors)
- agenda
- visited set

OPERATIONS COMBOS: RECURSION!:

```
def combinations(nums):
    if len(nums) < 2:
        return nums
    return {op(old, nums[-1]) for old in combinations(nums[:-1])}
```

OPERATIONS COMBO: RECURSION:

```
def combinations(nums)
    if len(nums) < 2:      ← set of nums
        return set(nums)
    out = set()
    for op in ops:
        out |= combinations([op(nums[0], nums[1])] + nums[2:])
```

*union (no dupes)

*applies op to first 2 elements then recursively looks for other combos

```
ops = [
    lambda x, y : x+y
    lambda x, y : x-y
    lambda x, y : x//y
    lambda x, y : x*y, ]
```

[1,2,3,4]:

combos([1,2,3])[_,3] etc.

[1,2] → op(1,2)

[1] → [1]

[...] gives brackets

HOW MANY STEPS CODE: BFS

```
def how-many steps(start, target, operations):
    agenda = [(start-num, 0)]
    visited = {start-num}
    while agenda:
        num, steps = agenda.pop(0)
        if num == target:
            return steps
        for f in operations:
            new_num = f(num)
            if new_num in visited:
                continue
            agenda.append(new-num, steps+1)
            visited.add(new-num)
```

F-STRINGS

```
print(f'the price
      is {price=}
      dollars )
```

ZIP:

```
x = [1,2]
y = [3,4]
zip(x,y) →
[(1,2), (3,4)]
```

ZIP ex:

```
def subtract_lists(L1, L2):
    return [i-j for i,j in zip(L1, L2)]
```

INFINITY:

```
shortest = float(''inf'')
longest = float(''-inf'')
```

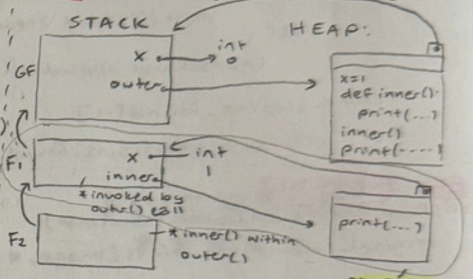
(try to find shortest/longest)

ENVIRONMENT DIAGRAMS:

```
n = 307
def f1():
    def f2():
        return n
    n = 308
    return f2
n = 309
def f3():
    n = 310
    f5 = f1()
    return [f5(), f4(), n]
def f4():
    return n
n = 311
print(f3())  * invokes F1
```

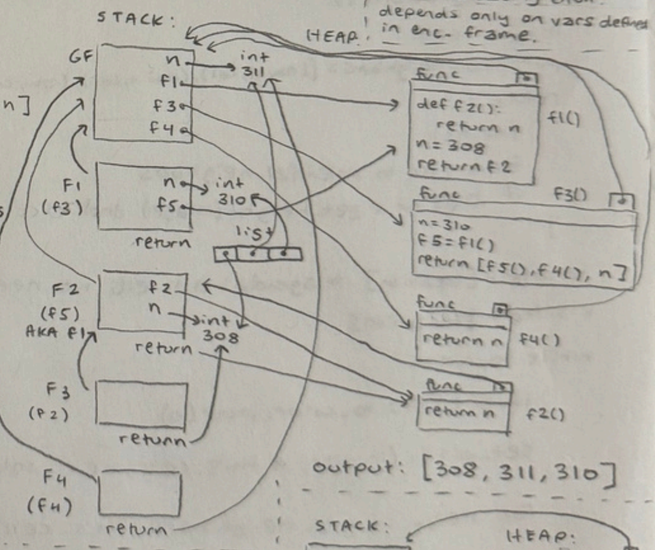
ENVIRONMENT DIAGRAMS:

```
x = 0
def outer():
    x = 1
    def inner():
        print('inner', x)
    inner()
    print('outer', x)
print('global', x)
outer()
inner()  ← produces NameError
print('global': x)
```



CLOSURE

(function/enclosing envir. depends only on vars defined in enc. frame.



output: [308, 311, 310]

SHORT-HAND NOTATION:

ex: [EXPRESSION for VAR in ITERABLE if COND]

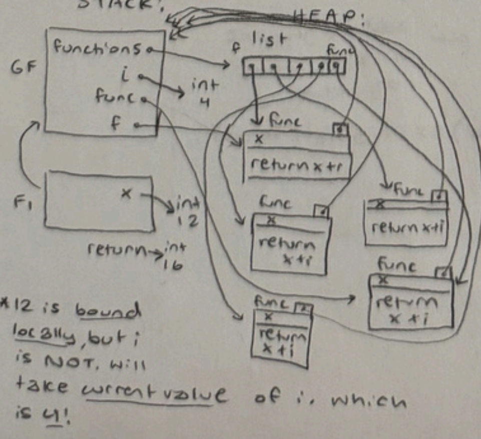
```
out = [num*2 for num in L]
```

ex: [(EXPRESSION if COND else OTHER_EXP) for VAR in ITERABLE]

ENVIRONMENT DIAGRAM: FUNCTIONS

```
functions = []
for i in range(5):
    def func(x):
        return x+i
    functions.append(func)
for f in functions:
    print(f(12))
```

→ prints 16 five times



*12 is bound locally, but i is NOT, will take current value of i, which is 4!

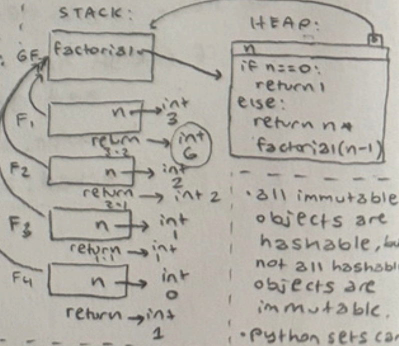
SPECIAL NOTES: shallow copy

- reverse() is shallow copy (changes original)
- use reversed() or [::-1] instead!
- * .copy() is shallow copy too!

ENVIRONMENT DIAGRAMS: RECURSION!

```
def factorial(n):
    if n == 0:
        return 1
    else:
        return (n *
            factorial(n-1))
factorial(3)
```

fac(3)
3 × 2
fac(2)
2 × 1
fac(1)
1 × _ ← 1
fac(0)



SPECIAL NOTES: immutable / hashable:

- all immutable objects are hashable, but not all hashable objects are immutable.
- Python sets can only include hashable objects

GRAPHS:

- vertices/nodes & edges/links [sometimes have direction]
  - ex: Ⓐ⟷Ⓑ
- tree structure → type of graph
  - has root/leaves
  - root = one thing as "center"
  - typically ONE WAY
  - ex: fam trees

COMPACT VERSION:

```
def longest-chain(database, first-name, last-name):
    chains-found = [[(first-name, last-name)]]
    for chains in chains-found: #[(-,-),(-,-)]
        last-link = chains[-1]  #(-, -)
        next-links = get-links(database, last-link[-1])
        for new-link in next-links:
            if new-link not in chain:
                new-chain = chain + [new-link]
                chains-found. append(new-chain)
    return chains-found[-1]
```

LONGEST CHAIN: **BFS**:

```
def longest_chain(database, first-name, last-name):
    chains-found =[[(first-name, last-name)]]
    agenda= chains-found.copy() # or [:]
      while agenda:
            chain = agenda.pop(0)  # [(-,-), (-,-)]
            last_link= chain[-1]   # (_,-)
            print(f'{last_link=}')
                                                  # or -1
            next-links = get_links(database, last-link[1])
            if next-links:                        # last-name
                for new-link in next-links:
                    if new_link not in chain:
                        new.chain = chain.copy()
                        new-chain. append(new-link)
                        agenda. append (new-chain)
                        print(f'added {new-chain}')
            else:
                chains-found. append(chain)
      return chains-found[-1]
      # or: return max(chains-found, key=len)
```

get-links: func returning list of tuples from ppl in database w/ same first names as name parameter

BFS: FLOOD FILL

unpacking * location[0], location[1]

```
def flood-fill(image, location, new-color):
    original-color = get-pixel(image, *location)

    def get-neighbors(cell):
        row, col = cell
        potential-neighbors = [(row+1, col), (row -1, col), (row, col+1), (row, col-1)]
        return [
            (nr, nc)
            for nr, nc in potential-neighbors
            if 0<= nr < get-height(image) and 0<= nr < get-width(image)
        ]

    to-color = [location] # agenda: all cells we need to color
    visited = {location}
    while to-color:
        this-cell = to-color. pop(0)
        set-pixel(image, * this-cell, new-color)

        for neigh in get-neighbors(this-cell):
            if (neigh not in visited
                    and get-pixel(image, *neigh) == original-color):
                to-color. append(neigh)
                visited. add (neigh)
```

add neighbors if it matches og color

RECURSION: lowest cost

optional param

```
def lowest-cost(recipes-db, food-name, forbidden-iter = None):
compound-recipes = ---
atomic-recipes = ...
    def calc-lowest-cost (food-name):
        if forbidden-iter and food-name in forbidden-iter:
            return None
        food-price-options = []
        if food-name in atomic-recipes:
            return atomic-recipes [food-name]
        elif food-name in compound-recipes:
            for ingred-list in compound-recipes [food-name]:
                price = 0
                for ingred-name, quant in ingred-list:
                    ingred-cost = calc-lowest-cost (ingred-name)
                    if ingred-cost is None:
                        price = None
                        break
                    price += quantity * ingred-cost
                if ingrediemt-cost:
                    food-price-options. append(price)
        elif food. name not in atomic-rec and ---- comp:
            return None
        return min(food-price-options)
    return calc-lowest-cost (food-name)
```

ATHERINE TU - 6.101 FINAL

INHERITANCE classes & lists

*does NOT modify class var

```
class A:
  foo = [1]
  def update(self, i):
     self.foo = self.foo + [i]
a = A()
a.update(10)
print (a.foo, A.foo)
OUT → [1, 10] [1]
```

*MODIFIES CLASS VAR!

AKA: self.foo.extend(i)

```
class A:
   foo = [1]
   def update(self, i):
      self.foo += [i]
a = A()
a.update(10)
print (a.foo, A.foo)
OUT → [1, 10] [1, 10]
```

(PARTIAL) COMBINATIONS list of lists, recursion

ex: combos([1,2])

out: [[], [1], [1,2], [2]]

```
def combos(inp):
  #base case
  if len(inp) == 0:
     return [[]]
  clist = []
  #recursive
  for combo in combos:
     clist.append(combo)
     clist.append( [inp[0]] + combo)
    #AKA  inp[0:1] + combo
          makes a list
```

generator version:

```
def combos.gen(inp):
   #base case
   if len(inp) == 0:
      yield []
      return
   for combo in combos.gen(inp[1:]):
      yield combo
      yield [inp[0]] + combo
```

ROUND ROBIN - GENERATOR

Ex: gen1 = iter([1,2,3])
gen2: iter([4,5])
gen3 = iter([7])
gens = [gen1, 2, 3]
out: 1, 4, 7, 2, 5, 3

```
def rr_gen(gens):        *tracks w/ bool
   done = [False for g in gens]
   while not all (done):
      for ix, g in enumerate(gens):
         if done [ix]:
            continue              *gets 1
         try:                      value
            yield next(g)          from each
         except StopIteration:     gen over
            done[ix] = True        & over!
```

GENERATOR: combine lists

```
def combine_gen(gens):
   for g in gens:
      yield from g
#yields each element 1 by 1 & combin
```

digits = "234" key_let("2") = ABC
combos: A.. B.. C.. ← 3 = DEF
4 = GHI

COMBINATIONS - recursion, limited # combo (3)

starts w/ 1st list

```
def let_from_dig(digits):
   if len(digits) == 0:
      yield ''
   for letter in key.letters[digits[0]]:
      for subword in let_from_dig(digits[1:]):
         yield letter + subword
```

GENERATOR - slicing

```
def gen_slice(iterable, start, stop, step):
  nextix = start
  for ix, elt in enumerate(iterable):
    if ix >= stop:
      break
    elif ix == nextix:
      yield elt
      nextix += step
```

COMBINATIONS - dictionary, unique

ppl: dict. of ppl & dishes they willing to bring

out: dict of a food mapped to 1 person (each food & person unique)

```
def potluck(ppl):
  def helper(ppl):
     if not ppl:
        return [{}]
     person = next(iter(ppl))
     rest = {k:v for k,v in ppl.items() if k != person}
     other_assign = helper(rest)
     result = []
     for assign in other_assign:
        for food in ppl[person]:
           if food not in assign:
              new_assign = assign.copy()
              new_assign[food] = person
              result.append(new_assign)
     return result
  return helper(ppl)
```

PATH-FINDING DFS/BFS

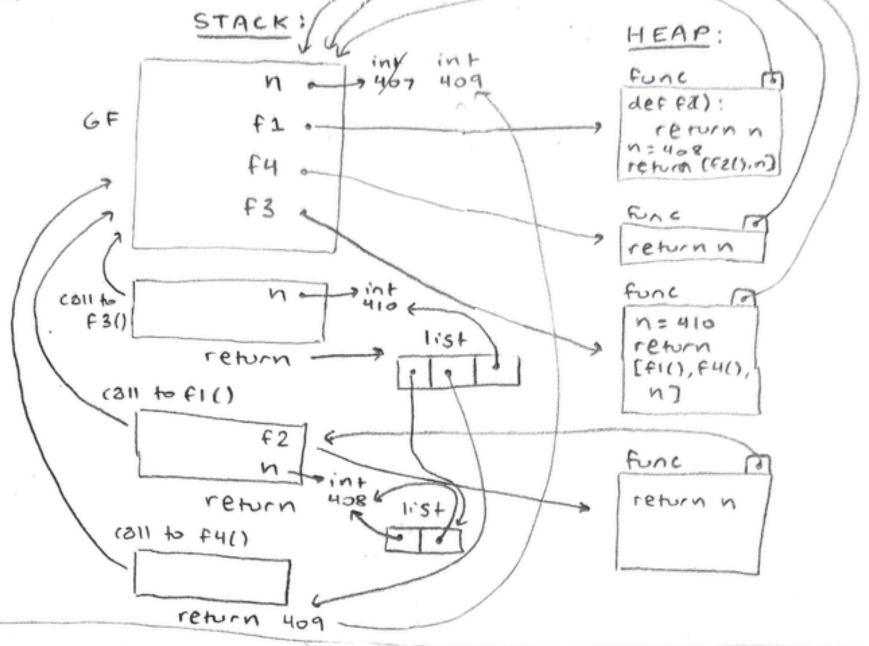
```
def find_path(neighbors_func, start_state, goal, bfs=True
   if goal_test(start_state):
      return (start_state,)
   agenda = [(start_state,)]
   visited = {start_state}
   while agenda:
      this_path = agenda.pop (0 if bfs else -1)
      terminal_state = this_path [-1] ← extracts last path
      for neighbor_state in neighbors_func (terminalstate)
         if neighbor_state not in visited:
            new_path = this_path + (neighbor_state,)
            if goal_test(neighbor_state):
               return new_path
            agenda.append (new_path)
            visited.add (neighbor_state)
```

OPERATIONS COMBINATION

```
def combos(nums)
  if len(nums) < 2:
     return set(nums)
  out = set()
  for op in ops:
     out 1 = combos([op(nums[0], nums[1])]
                   + nums[2:])
```

ENVIRONMENT DIAGRAM - Scope, functions

```
n=407
def f1():
  def f2():
    return n
  n=408
  return [f2(), n]
def f4():
  return n
n=409
def f3():
  n=410
  return [f1(), f4(), n]
print(f3())
OUT→ [[408,408], 409, 410]
```



*if no n in frame, go to enclosing frame (GF)

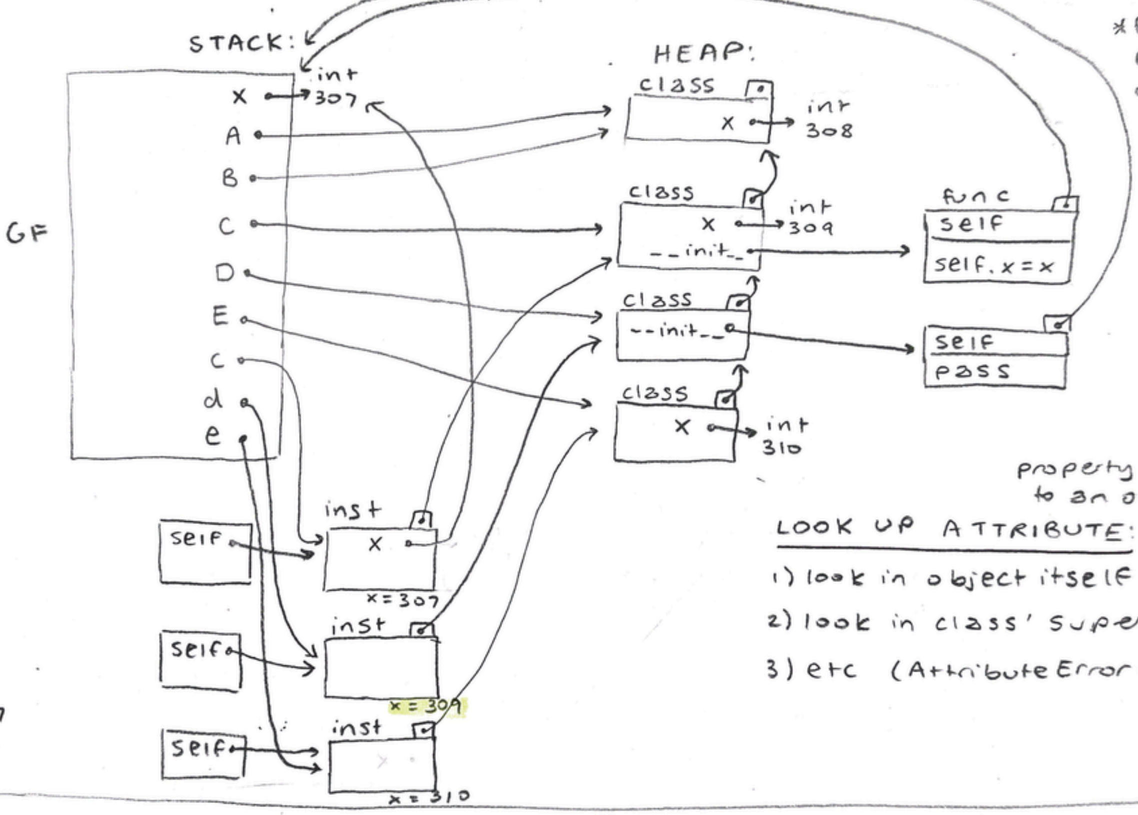
LOOK UP VAR:

1) look in current fra
2) look in parent framp
3) look at GF
4) Name Error if N/A

ENVIRONMENT DIAGRAM - classes, inheritance

```
x=307
class A:        *class
  x=308           attribute
B=A
class C(B):
  x=309
  def __init__(self):
    self.x = x
class D(C):
  def __init__(self):
    pass
class E(D):
  x=310
c=C()
d=D()
e=E()
print(c.x)              307
print(d.x)     OUT→     309
print(e.x)              310
```

*funcs go t
enc. fram
of class
(in this
case → G



property belonging to an obj (class)

LOOK UP ATTRIBUTE:

1) look in object itself
2) look in class' superclass
3) etc (AttributeError is N/A)