Midterm 3 Review

Announcements

Prefixes

```
def prefixes(s):
    """Return a list of all of the list prefixes of s.

    >>> prefixes([1, 2, 3])
    [[1], [1, 2], [1, 2, 3]]
    """

    so_far = []

    result = []

    for x in s:

        so_far.append(x)

        result.append(so_far)

    return result
```




```
>>> prefixes([1, 2, 3])
```


https://pythontutor.com/cp/composingprograms.html#code=def%20prefixes%28s%29%3A%0A%20%20%20%20so_far%20%3D%20%5B%5D%0A%20%20%20%20result%20%3D%20%5B%5D%0A%20%20%20%20for%20x%20in%20s%3A%0A%20%20%20%20%20%20%20%20so_far.append%28x%29%0A%20%20%20%20%20%20%20%20result.append%28so_far%29%0A%20%20%20%20return%20result%0A%0Aprint%28prefixes%28%5B1,%202,%203%5D%29%29&cumulative=true&curInstr=0&mode=display&origin=composingprograms.js&py=3&rawInputLstJSON=%5B%5D

Prefixes

```
def prefixes(s):
    """Return a list of all of the list prefixes of s.

    >>> prefixes([1, 2, 3])
    [[1], [1, 2], [1, 2, 3]]
    """

    so_far = []

    result = []

    for x in s:

        so_far.append(x)
                      list(so_far)
        result.append(so_far)

    return result
```

```
>>> prefixes([1, 2, 3])

[[1, 2, 3], [1, 2, 3], [1, 2, 3]]
```

How can we fix the implementation?

pollev.com/cs61a

https://pythontutor.com/cp/composingprograms.html#code=def%20prefixes%28s%29%3A%0A%20%20%20%20so_far%20%3D%20%5B%5D%0A%20%20%20%20result%20%3D%20%5B%5D%0A%20%20%20%20for%20x%20in%20s%3A%0A%20%20%20%20%20%20%20%20so_far.append%28x%29%0A%20%20%20%20%20%20%20%20result.append%28so_far%29%0A%20%20%20%20return%20result%0A%0Aprint%28prefixes%28%5B1,%202,%203%5D%29%29&cumulative=true&curInstr=0&mode=display&origin=composingprograms.js&py=3&rawInputLstJSON=%5B%5D

Recursion Recipe

- Write down an **example input**
- What **small initial choice** can I make?
- What **recursive call for each option**?
  - Write down the recursive call(s) for your example input
  - Write down what each of those calls returns for your example input

Tree Recursion Practice

Spring 2023 Midterm 2 Question 4(a)

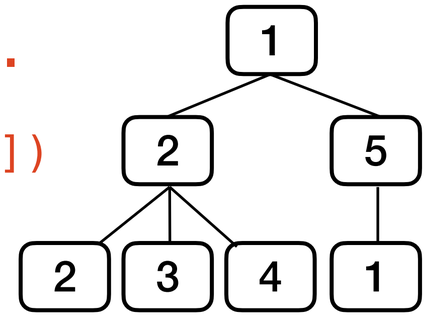
Implement exclude, which takes a Tree t and a value x. It returns a Tree containing the root node of t as well as each non-root node of t with a label not equal to x. The parent of a node in the result is its nearest ancestor node that is not excluded.

```
def exclude(t, x):
    """Return a tree with the non-root nodes of tree t labeled anything but x.

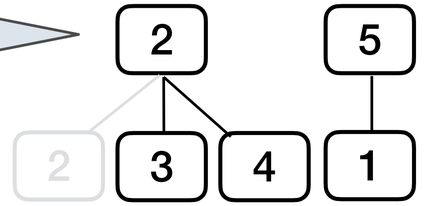
    >>> t = Tree(1, [Tree(2, [Tree(2), Tree(3), Tree(4)]), Tree(5, [Tree(1)])])
    >>> print(exclude(t, 2))
    1
      3
      4
      5
        1
    """
    filtered_branches = map(lambda y: _____
    bs = []
    for b in filtered_branches:

        if _________________:

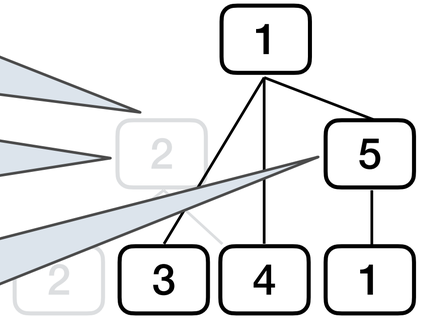
            bs._________(_____________)
        else:
            bs.append(b)
    return Tree(t.label, bs)
```



What will the recursive call on each branch return?



What should we do with those return values?



Branch has label x?
Take its branches

Otherwise we're cool with the branch as-is

Spring 2023 Midterm 2 Question 4(a)

Implement exclude, which takes a Tree t and a value x. It returns a Tree containing the root node of t as well as each non-root node of t with a label not equal to x. The parent of a node in the result is its nearest ancestor node that is not excluded.

```
def exclude(t, x):
    """Return a tree with the non-root nodes of tree t labeled anything but x.

    >>> t = Tree(1, [Tree(2, [Tree(2), Tree(3), Tree(4)]), Tree(5, [Tree(1)])])
    >>> print(exclude(t, 2))
    1
      3
      4
      5
        1
    """
    filtered_branches = map(lambda y: _____, t.branches)
    bs = []
    for b in filtered_branches:
        if _____:
            bs._____(_____)
        else:
            bs.append(b)
    return Tree(t.label, bs)
```

exclude(y, x)

In Spring 2023, 20% of students got this right

b.label == x

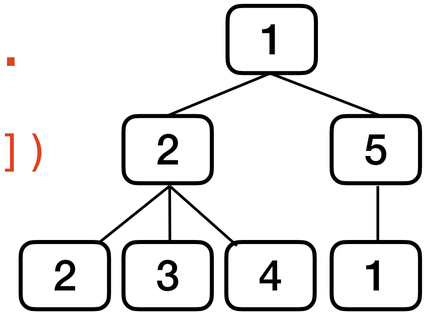
37% of students got this right

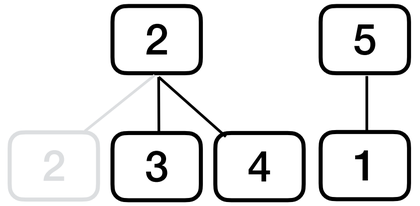
30% got it right; 1 of 4 options

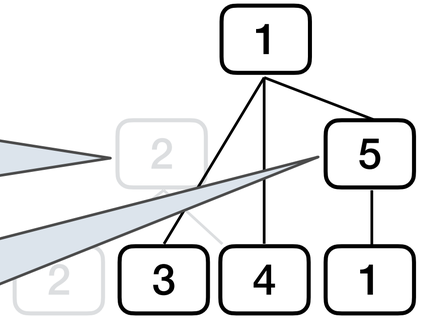
extend

b.branches

24% got it right







Branch has label x? Take its branches

Otherwise we're cool with the branch as-is

String Practice

Fruit: Warm-Up

```
def display(fruit: str, count: int) -> str:
    """Display a count of a fruit in angle brackets. When count is 1, the fruit name should
    be singular: drop the final s, so that display('apples', 1) returns '<1 apple>'. Assume
    that every plural fruit name can be made singular by removing the last letter.

    >>> display('apples', 3)
    '<3 apples>'
    >>> display('apples', 1)
    '<1 apple>'
    >>> display('kiwis', 12)
    '<12 kiwis>'
    >>> print(display('apples', 3) + display('kiwis', 3))
    <3 apples><3 kiwis>
    """
    assert count >= 1 and fruit[-1] == 's'
    if count == 1:
        ________________
    return f'_______________'
```

Fruit: Warm-Up

```
def display(fruit: str, count: int) -> str:
    """Display a count of a fruit in angle brackets. When count is 1, the fruit name should
    be singular: drop the final s, so that display('apples', 1) returns '<1 apple>'. Assume
    that every plural fruit name can be made singular by removing the last letter.

    >>> display('apples', 3)
    '<3 apples>'
    >>> display('apples', 1)
    '<1 apple>'
    >>> display('kiwis', 12)
    '<12 kiwis>'
    >>> print(display('apples', 3) + display('kiwis', 3))
    <3 apples><3 kiwis>
    """
    assert count >= 1 and fruit[-1] == 's'
    if count == 1:
        fruit = fruit[:-1]
    return f'<{count} {fruit}>'
```

Dictionary/Recursion Practice

Buying Fruit

```
def buy(fruits_to_buy: list[str], prices: dict[str, int], total_amount: int) -> None:
    """Print ways to buy some of each fruit so that
    the sum of prices is amount.

    >>> prices = {'oranges': 4, 'apples': 3,
                  'bananas': 2, 'kiwis': 9}
    >>> buy(['apples', 'oranges', 'bananas'],
            prices, 16)
    <2 apples><1 orange><3 bananas>
    <2 apples><2 oranges><1 banana>
    """
```

Implement the buy function that takes three parameters:

fruits_to_buy: A list of strings representing the fruits you need to buy. *At least one of each fruit must be bought.*
prices: A dictionary where the keys are fruit names (strings) and the values are positive integers representing the cost of each fruit.
total_amount: An integer representing the total money available for purchasing the fruits.

The function should print all possible ways to buy the required fruits so that the combined cost equals total_amount. You can only select fruits mentioned in the fruits_to_buy list.

Buying Fruit

```
def buy(fruits_to_buy: list[str], prices: dict[str, int], total_amount: int) -> None:
    """Print ways to buy some of each fruit so that
    the sum of prices is amount.

    >>> prices = {'oranges': 4, 'apples': 3,
                  'bananas': 2, 'kiwis': 9}
    >>> buy(['apples', 'oranges', 'bananas'],
            prices, 16)
    <2 apples><1 orange><3 bananas>
    <2 apples><2 oranges><1 banana>
    """
    def add(fruits: list[str], amount: int, cart: str) -> None:
        if fruits == [] and amount == 0:
            print(cart)
        elif fruits and amount > 0:
            fruit = fruits[0]
            price = _____________
            for k in _____________________________:
                add(__________, ___________________, ____________________________)
    add(fruits_to_buy, total_amount, '')
```

The display function will help you add fruit to the cart

```
def display(fruit: str, count: int) -> str:
```

Buying Fruit

```
def buy(fruits_to_buy: list[str], prices: dict[str, int], total_amount: int) -> None:
    """Print ways to buy some of each fruit so that
    the sum of prices is amount.

    >>> prices = {'oranges': 4, 'apples': 3,
                  'bananas': 2, 'kiwis': 9}
    >>> buy(['apples', 'oranges', 'bananas'],
            prices, 16)
    <2 apples><1 orange><3 bananas>
    <2 apples><2 oranges><1 banana>
    """
    def add(fruits: list[str], amount: int, cart: str) -> None:
        if fruits == [] and amount == 0:
            print(cart)
        elif fruits and amount > 0:
            fruit = fruits[0]
            price = prices[fruit]
            for k in range(1, amount // price + 1):
                add(fruits[1:], amount - price * k, cart + display(fruit, k))
    add(fruits_to_buy, total_amount, '')
```

The display function will help you add fruit to the cart

```
def display(fruit: str, count: int) -> str:
```

Class Practice

Mic and Speakers (Spring 2024 Midterm)

```
class Mic:
    """A microphone connected to speakers.

    >>> m = Mic()  # Front is connected automatically
    >>> m.sing('Is this thing on?')
    Front - Is this thing on?
    >>> Speaker(str.lower).connect(m, 'Left Side')
    >>> m.sing("You BELONG with me.")
    Front - You BELONG with me.
    Left Side - you belong with me.
    """
    def __init__(self):
        self.speakers = _______

    def sing(self, lyrics: str):
        for k in self.speakers.keys():
            print(k, '-', _________________.repeat(lyrics))

class Speaker:
    def __init__(self, transform):
        self.transform = transform

    def connect(self, m: Mic, location: str):
        _______

    def repeat(self, s: str) -> str:
        return _______
```

If you sing lyrics into a mic, every connected speaker repeats them.

A **Mic** instance has a dictionary **speakers** containing **Speaker** instances as values, each with its **location** (**str**) as its key. Its **sing** method takes a string **lyrics** and invokes the **repeat** method of each **Speaker** instance connected to it.

A **Speaker** takes a **transform** function that takes and returns a string. To **connect** a **Speaker** instance to **m** (**Mic**) in a **location** (**str**), add that instance to the **speakers** dictionary of **m** in that **location.** To **repeat** a signal **s** (**str**), return the result of calling the speaker's **transform** function on **s.**

Every **Mic** starts connected to a **Speaker** in the **Front** location that repeats the exact same signal it receives.

Mic and Speakers (Spring 2024 Midterm)

```
class Mic:
    """A microphone connected to speakers.

    >>> m = Mic()  # Front is connected automatically
    >>> m.sing('Is this thing on?')
    Front - Is this thing on?
    >>> Speaker(str.lower).connect(m, 'Left Side')
    >>> m.sing("You BELONG with me.")
    Front - You BELONG with me.
    Left Side - you belong with me.
    """
    def __init__(self):
        self.speakers = {'Front': Speaker(lambda x: x)}

    def sing(self, lyrics: str):
        for k in self.speakers.keys():
            print(k, '-', self.speakers[k].repeat(lyrics))

class Speaker:
    def __init__(self, transform):
        self.transform = transform

    def connect(self, m: Mic, location: str):
        m.speakers[location] = self

    def repeat(self, s: str) -> str:
        return self.transform(s)
```

If you sing lyrics into a mic, every connected speaker repeats them.

A **Mic** instance has a dictionary **speakers** containing **Speaker** instances as values, each with its **location** (**str**) as its key. Its **sing** method takes a string **lyrics** and invokes the **repeat** method of each **Speaker** instance connected to it.

A **Speaker** takes a **transform** function that takes and returns a string. To **connect** a **Speaker** instance to **m** (**Mic**) in a **location** (**str**), add that instance to the **speakers** dictionary of **m** in that **location.** To **repeat** a signal **s** (**str**), return the result of calling the speaker's **transform** function on **s.**

Every **Mic** starts connected to a **Speaker** in the **Front** location that repeats the exact same signal it receives.

Example: Three Attributes

```
class A:
    x, y, z = 0, 1, 2

    def f(self):
        return [self.x, self.y, self.z]

class B(A):
    """What would Python Do?

    >>> A().f()

    [0, 1, 2]

    >>> B().f()
     [6, 1, 'A']
    ______________

    """
    x = 6
    def __init__(self):
        self.z = 'A'
```

A class

x: 0
y: 1
z: 2

B class

x: 6

A instance

B instance

z: 'A'

Class Attributes

A class attribute can be accessed from either an instance or its class. There is only one value for a class attribute, regardless of how many instances.

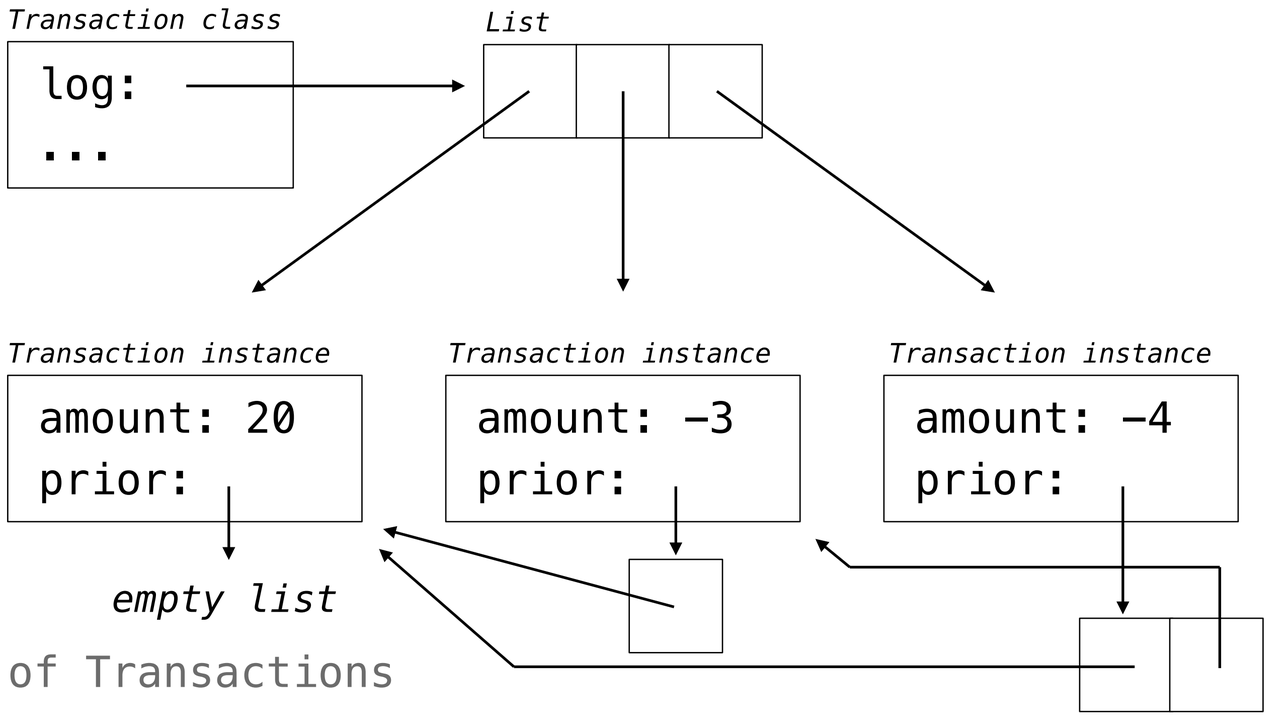
```
class Transaction:
    """A logged transaction.

    >>> s = [20, -3, -4]
    >>> ts = [Transaction(x) for x in s]
    >>> ts[1].balance()
    17
    >>> ts[2].balance()
    13
    """
    log = []

    def __init__(self, amount):
        self.amount = amount
        self.prior = list(self.log)  # a list of Transactions
        self.log.append(self)

    def balance(self):
        """The sum of amounts for this transaction and all prior transactions"""
        return self.amount + sum( [t.amount for t in self.prior] )
```

Always bound to some Transaction instance

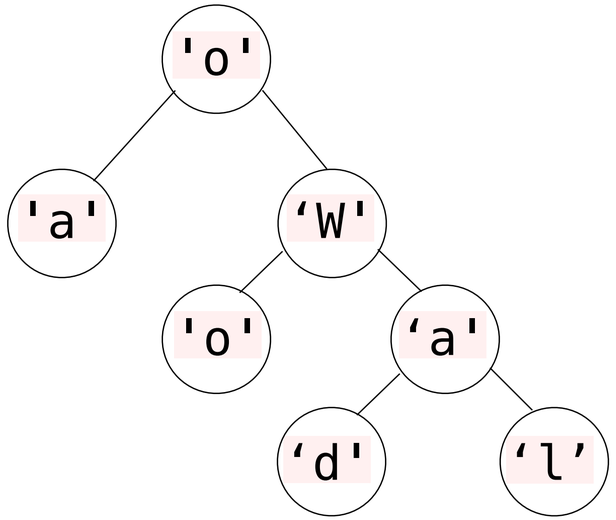


Equivalently: list(type(self).log) **or** list(Transaction.log)

Generator Practice

Generator Where's Waldo

```
def all_elements(t: Tree):
    for b in t.branches:
        yield from all_elements(b)
    yield t.label
```




```
>>> t = Tree('o', [Tree('a'), Tree('W', [Tree('o'), Tree('a', [Tree('d'), Tree('l')])])])
>>> a = all_elements(t)
>>> a
<generator object all_elements at 0x104e4ddd0>
>>> next(a)
'a'

>>> list(reversed(list(a)[:-1]))
['W', 'a', 'l', 'd', 'o']
```

Write an expression that uses **a** and evaluates to ['W', 'a', 'l', 'd', 'o']

More List Practice

Lists of Fibonacci Numbers Recursively

List mutations can be used to build up a list incrementally, even using recursion

```
def fibs_rec(n):
    """Return a list of the first n fibonacci numbers.

    >>> fibs_rec(10)
    [0, 1, 1, 2, 3, 5, 8, 13, 21, 34]
    """
    s = []
    def f(i):
        _________:
            if i < 2:
                s.append(i)
            else:
                s.append(s[-2] + s[-1])
            __________
    _____
    return s
```

Lists of Fibonacci Numbers Recursively

List mutations can be used to build up a list incrementally, even using recursion

```
def fibs_rec(n):
    """Return a list of the first n fibonacci numbers.

    >>> fibs_rec(10)
    [0, 1, 1, 2, 3, 5, 8, 13, 21, 34]
    """
    s = []
    def f(i):
        if i < n:
            if i < 2:
                s.append(i)
            else:
                s.append(s[-2] + s[-1])
            f(i + 1)
    f(0)
    return s
```