DATA C88C
Spring 2025

Final Review
Discussion 12: April 29, 2025

Environment Diagrams

Q1: Nested Calls Diagrams

Draw the environment diagram that results from executing the code below.

See the web version of this resource for the environment diagram.

HOFs

Q2: Make Repeater

Implement the function `make_repeater` so that `make_repeater(f, n)(x)` returns `f(f(...f(x)...))`, where `f` is applied `n` times. That is, `make_repeater(f, n)` returns another function that can then be applied to another argument. For example, `make_repeater(square, 3)(42)` evaluates to `square(square(square(42)))`.

```
def make_repeater(f, n):
    """Returns the function that computes the nth application of f.
    >>> add_one = lambda x: x + 1
    >>> add_three = make_repeater(add_one, 3)
    >>> add_three(5)
    8
    >>> make_repeater(triple, 5)(1) # 3 * 3 * 3 * 3 * 3 * 1
    243
    >>> make_repeater(square, 2)(5) # square(square(5))
    625
    >>> make_repeater(square, 4)(5) # square(square(square(square(5))))
    152587890625
    >>> make_repeater(square, 0)(5) # Yes, it makes sense to apply the function zero
    times!
    5
    """
    g = lambda x: x
    while n > 0:
        g = composer(f, g)
        n = n - 1
    return g

def make_repeater2(f, n): # Alternative solution
    def inner_func(x):
        k = 0
        while k < n:
            x, k = f(x), k + 1
        return x
    return inner_func

def composer(func1, func2):
    """Returns a function f, such that f(x) = func1(func2(x))."""
    def f(x):
        return func1(func2(x))
    return f
```

Solution using `composer`:

We create a new function in every iteration of the `while` statement by calling `composer`.

Solution not using `composer`:

We create a single inner function that contains the `while` logic needed to do calculations directly, as opposed to creating another function for every `while` loop iteration.

Recursion

Q3: Subsequences

A subsequence of a sequence `S` is a subset of elements from `S`, in the same order they appear in `S`. Consider the list `[1, 2, 3]`. Here are a few of its subsequences `[]`, `[1, 3]`, `[2]`, and `[1, 2, 3]`.

Write a function that takes in a list and returns all possible subsequences of that list. The subsequences should be returned as a list of lists, where each nested list is a subsequence of the original input.

In order to accomplish this, you might first want to write a function `insert_into_all` that takes an item and a list of lists, adds the item to the beginning of each nested list, and returns the resulting list.

```
def insert_into_all(item, nested_list):
    """Return a new list consisting of all the lists in nested_list,
    but with item added to the front of each. You can assume that
    nested_list is a list of lists.

    >>> nl = [[], [1, 2], [3]]
    >>> insert_into_all(0, nl)
    [[0], [0, 1, 2], [0, 3]]
    """
    return [[item] + lst for lst in nested_list]

def subseqs(s):
    """Return a nested list (a list of lists) of all subsequences of S.
    The subsequences can appear in any order. You can assume S is a list.

    >>> seqs = subseqs([1, 2, 3])
    >>> sorted(seqs)
    [[], [1], [1, 2], [1, 2, 3], [1, 3], [2], [2, 3], [3]]
    >>> subseqs([])
    [[]]
    """
    if not s:
        return [[]]
    else:
        subset = subseqs(s[1:])
        return insert_into_all(s[0], subset) + subset
```

OOP

Q4: Bear

Implement the `SleepyBear`, and `WinkingBear` classes so that calling their `print` method matches the doctests. Use as little code as possible and try not to repeat any logic from `Eye` or `Bear`. Each blank can be filled with just two short lines.

Discussion Time: Before writing code, talk about what is different about a `SleepyBear` and a `Bear`. When using inheritance, you only need to implement the differences between the base class and subclass. Then, talk about

what is different about a `WinkingBear` and a `Bear`. Can you think of a way to make the bear wink without a new implementation of `print`?

class Eye: """An eye.

```
    >>> Eye().draw()
    '0'
    >>> print(Eye(False).draw(), Eye(True).draw())
    0 -
    """
    def __init__(self, closed=False):
        self.closed = closed

    def draw(self):
        if self.closed:
            return '-'
        else:
            return '0'

class Bear:
    """A bear.

    >>> Bear().print()
    ? 0o0?
    """
    def __init__(self):
        self.nose_and_mouth = 'o'

    def next_eye(self):
        return Eye()

    def print(self):
        left, right = self.next_eye(), self.next_eye()
        print('? ' + left.draw() + self.nose_and_mouth + right.draw() + '?')
```

```
class SleepyBear(Bear):
    """A bear with closed eyes.

    >>> SleepyBear().print()
    ? -o-?
    """
    def next_eye(self):
        return Eye(True)

class WinkingBear(Bear):
    """A bear whose left eye is different from its right eye.

    >>> WinkingBear().print()
    ? -o0?
    """
    def __init__(self):
        super().__init__()
        self.eye_calls = 0

    def next_eye(self):
        self.eye_calls += 1
        return Eye(self.eye_calls % 2)
```

Linked Lists

A linked list is a `Link` object or `Link.empty`.

You can mutate a `Link` object `s` in two ways: - Change the first element with `s.first = ...` - Change the rest of the elements with `s.rest = ...`

You can make a new `Link` object by calling `Link`: - `Link(4)` makes a linked list of length 1 containing 4. - `Link(4, s)` makes a linked list that starts with 4 followed by the elements of linked list `s`.

```
class Link:
    """A linked list is either a Link object or Link.empty

    >>> s = Link(3, Link(4, Link(5)))
    >>> s.rest
    Link(4, Link(5))
    >>> s.rest.rest.rest is Link.empty
    True
    >>> s.rest.first * 2
    8
    >>> print(s)
    <3 4 5>
    """
    empty = ()

    def __init__(self, first, rest=empty):
        assert rest is Link.empty or isinstance(rest, Link)
        self.first = first
        self.rest = rest

    def __repr__(self):
        if self.rest:
            rest_repr = ', ' + repr(self.rest)
        else:
            rest_repr = ''
        return 'Link(' + repr(self.first) + rest_repr + ')'

    def __str__(self):
        string = '<'
        while self.rest is not Link.empty:
            string += str(self.first) + ' '
            self = self.rest
        return string + str(self.first) + '>'
```

Q5: Linear Sublists

Definition: A *sublist* of linked list `s` is a linked list of some of the elements of `s` in order. For example, `<3 6 2 5 1 7>` has sublists `<3 2 1>` and `<6 2 7>` but not `<5 6 7>`.

Definition: A *linear sublist* of a linked list of numbers `s` is a sublist in which the difference between adjacent numbers is always the same. For example `<2 4 6 8>` is a linear sublist of `<1 2 3 4 6 9 1 8 5>` because the difference between each pair of adjacent elements is 2.

Implement `linear` which takes a linked list of numbers `s` (either a `Link` instance or `Link.empty`). It returns the longest linear sublist of `s`. If two linear sublists are tied for the longest, return either one.

```
def linear(s):
    """Return the longest linear sublist of a linked list s.

    >>> s = Link(9, Link(4, Link(6, Link(7, Link(8, Link(10))))))
    >>> linear(s)
    Link(4, Link(6, Link(8, Link(10))))
    >>> linear(Link(4, Link(5, s)))
    Link(4, Link(5, Link(6, Link(7, Link(8)))))
    >>> linear(Link(4, Link(5, Link(4, Link(7, Link(3, Link(2, Link(8))))))))
    Link(5, Link(4, Link(3, Link(2))))
    """
    def complete(first, rest):
        "The longest linear sublist of Link(first, rest) with difference d."
        if rest is Link.empty:
            return Link(first, rest)
        elif rest.first - first == d:
            return Link(first, complete(rest.first, rest.rest))
        else:
            return complete(first, rest.rest)
    if s is Link.empty:
        return s
    longest = Link(s.first) # The longest linear sublist found so far
    while s is not Link.empty:
        t = s.rest
        while t is not Link.empty:
            d = t.first - s.first
            candidate = Link(s.first, complete(t.first, t.rest))
            if length(candidate) > length(longest):
                longest = candidate
            t = t.rest
        s = s.rest
    return longest

def length(s):
    if s is Link.empty:
        return 0
    else:
        return 1 + length(s.rest)
```

There are three cases: - If `rest` is empty, return a one-element list containing just `first`. - If `rest.first` is in the linear sublist that starts with `first`, then build a list that contains `first`, and `rest.first`. - Otherwise, `complete(first, rest.rest)`.

This while loop is creating a `candidate` linear sublist for every two possible starting values: `s.first` and `t.first`. The rest of the linear sublist must be in `t.rest`.

Iterators

Q6: Repeated

Implement `repeated`, which takes in an iterator `t` and an integer `k` greater than 1. It returns the first value in `t` that appears `k` times in a row.

> **Important:** Call `next` on `t` only the minimum number of times required. Assume that there is an element of `t` repeated at least `k` times in a row.
>
> **Hint**: If you are receiving a `StopIteration` exception, your `repeated` function is calling `next` too many times.

```
def repeated(t, k):
    """Return the first value in iterator t that appears k times in a row,
    calling next on t as few times as possible.

    >>> s = iter([10, 9, 10, 9, 9, 10, 8, 8, 8, 7])
    >>> repeated(s, 2)
    9
    >>> t = iter([10, 9, 10, 9, 9, 10, 8, 8, 8, 7])
    >>> repeated(t, 3)
    8
    >>> u = iter([3, 2, 2, 2, 1, 2, 1, 4, 4, 5, 5, 5])
    >>> repeated(u, 3)
    2
    >>> repeated(u, 3)
    5
    >>> v = iter([4, 1, 6, 6, 7, 7, 8, 8, 2, 2, 2, 5])
    >>> repeated(v, 3)
    2
    """
    assert k > 1
    count = 0
    last_item = None
    while True:
        item = next(t)
        if item == last_item:
            count += 1
        else:
            last_item = item
            count = 1
        if count == k:
            return item
```

Trees

Q7: Long Paths

Implement `long_paths`, which returns a list of all *paths* in a tree with length at least `n`. A path in a tree is a list of node labels that starts with the root and ends at a leaf. Each subsequent element must be from a label of a branch of the previous value's node. The *length* of a path is the number of edges in the path (i.e. one less than the number of nodes in the path). Paths are ordered in the output list from left to right in the tree. See the doctests for some examples.

```
def long_paths(t, n):
    """Return a list of all paths in t with length at least n.

    >>> long_paths(Tree(1), 0)
    [[1]]
    >>> long_paths(Tree(1), 1)
    []
    >>> t = Tree(3, [Tree(4), Tree(4), Tree(5)])
    >>> left = Tree(1, [Tree(2), t])
    >>> mid = Tree(6, [Tree(7, [Tree(8)]), Tree(9)])
    >>> right = Tree(11, [Tree(12, [Tree(13, [Tree(14)])])])
    >>> whole = Tree(0, [left, Tree(13), mid, right])
    >>> print(whole)
    0
      1
        2
        3
          4
          4
          5
      13
      6
        7
          8
        9
      11
        12
          13
            14
    >>> for path in long_paths(whole, 2):
    ...     print(path)
    ...
    [0, 1, 2]
    [0, 1, 3, 4]
    [0, 1, 3, 4]
    [0, 1, 3, 5]
    [0, 6, 7, 8]
    [0, 6, 9]
    [0, 11, 12, 13, 14]
    >>> for path in long_paths(whole, 3):
    ...     print(path)
    ...
    [0, 1, 3, 4]
    [0, 1, 3, 4]
    [0, 1, 3, 5]
    [0, 6, 7, 8]
    [0, 11, 12, 13, 14]
    >>> long_paths(whole, 4)
    [[0, 11, 12, 13, 14]]
    """
    if n <= 0 and t.is_leaf():
```

Document the Occasion

Please all fill out the attendance form (one submission per person per week).