MUTABLE FUNCTIONS AND MIDTERM REVIEW 6

COMPUTER SCIENCE 61A

July 12, 2016

1 Nonlocal

Until now, you've been able to access variables in parent frames, but you have not been able to modify them. The `nonlocal` keyword can be used to modify a variable in the parent frame outside the current frame. For example, consider `stepper`, which uses `nonlocal` to modify `num`:

```
def stepper(num):
    def step():
        nonlocal num  # declares num as a nonlocal variable
        num = num + 1 # modifies num in the stepper frame
        return num
    return step
```

However, there are two important caveats with `nonlocal` variables:

- **Global variables** cannot be modified using the `nonlocal` keyword.
- **Variables in the current frame** cannot be overridden using the `nonlocal` keyword.

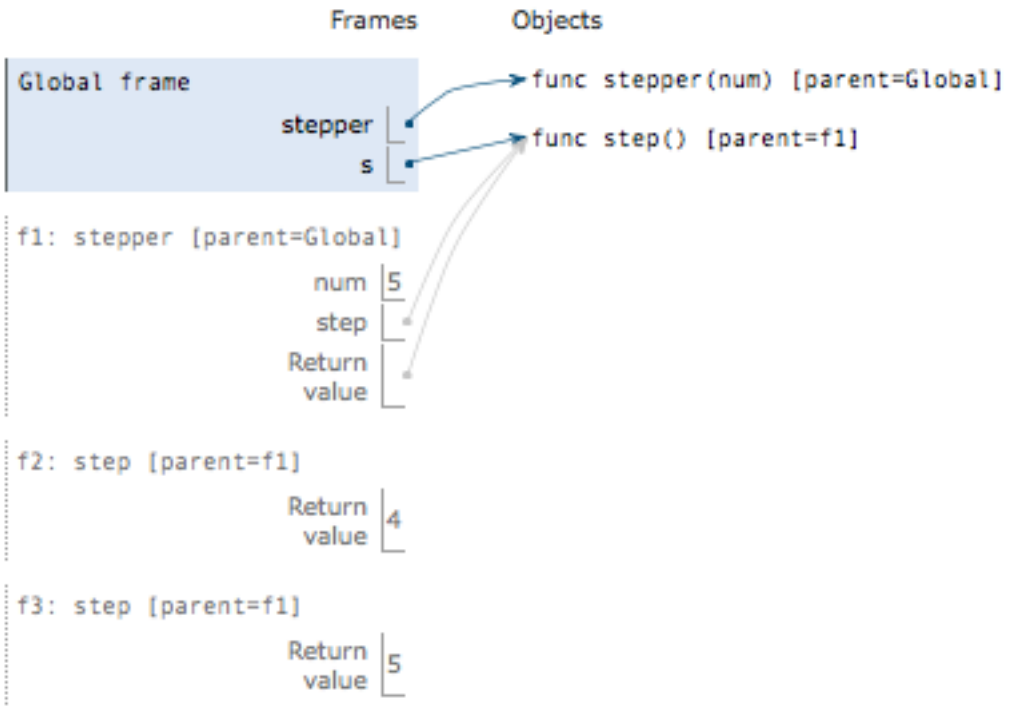
1.1 Environment Diagrams

1. Draw the environment diagram for the code below:

```
def stepper(num):
    def step():
        nonlocal num
        num = num + 1
        return num
    return step

s = stepper(3)
s()
s()
```

Solution:

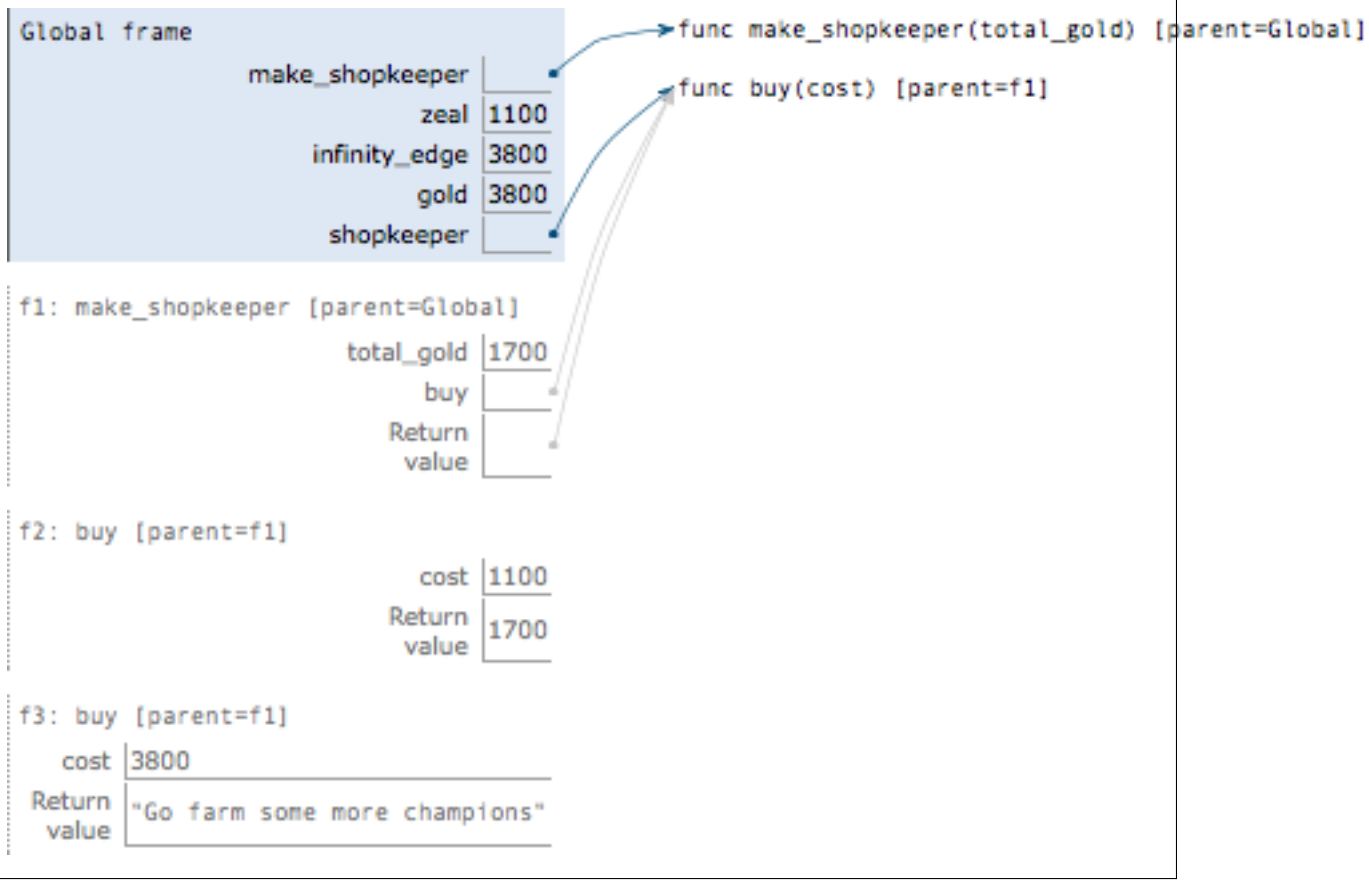


2. Given the definition of `make_shopkeeper` below, draw the environment diagram.

```
def make_shopkeeper(total_gold):
    def buy(cost):
        nonlocal total_gold
        if total_gold < cost:
            return 'Go farm some more champions'
        total_gold = total_gold - cost
        return total_gold
    return buy

infinity_edge, zeal, gold = 3800, 1100, 3800
shopkeeper = make_shopkeeper(gold - 1000)
shopkeeper(zeal)
shopkeeper(infinity_edge)
```

Solution:



1.2 Some Common Misconceptions

1. What is wrong with the following code?

```
a = 5
def another_add_one():
    nonlocal a
    a += 1
another_add_one()
```

Solution: Nonlocal cannot be used to modify variables in the global frame.

2. What is wrong with the following code?

```
def adder(x):
    def add(y):
        nonlocal x, y
        x += y
        return x
    return add
adder(2)(3)
```

Solution: There is no variable y defined in a parent frame, and y is already a local variable.

1.3 Fill in the Blank

1. The bathtub below simulates an epic battle between Finn and Kylo Ren over a populace of rubber duckies. Fill in the body of `ducky` so that all doctests pass.

```
def bathtub(n):
    """
    >>> annihilator = bathtub(500) # the force awakens...
    >>> kylo_ren = annihilator(10)
    >>> kylo_ren()
    490 rubber duckies left
    >>> finn = annihilator(-20)
    >>> finn()
    510 rubber duckies left
    >>> kylo_ren()
    500 rubber duckies left
    """
    def ducky_annihilator(rate):
        def ducky():
```

Solution:

```
            nonlocal n
            n = n - rate
            print(n, 'rubber duckies left')
```

```
        return ducky
    return ducky_annihilator
```

2 Midterm Review

2.1 Environment Diagrams

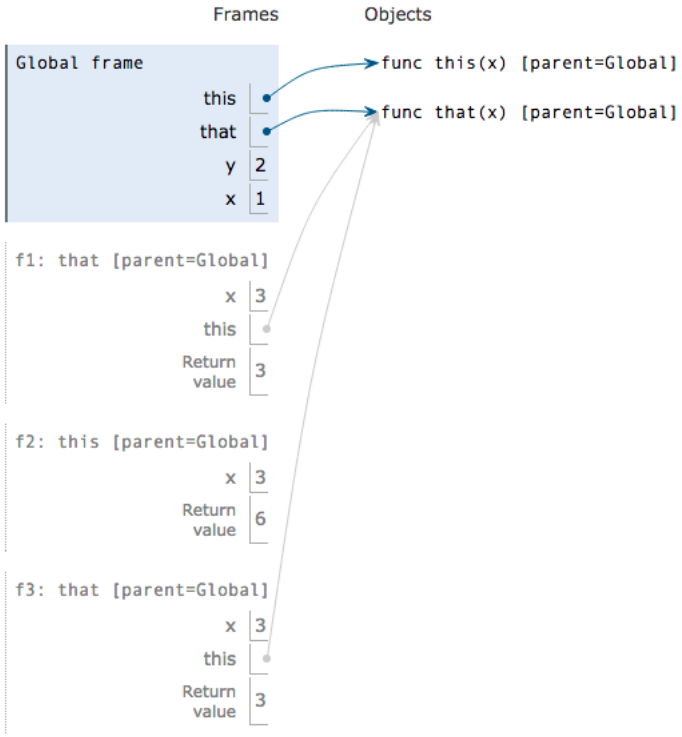
1. Draw the environment diagram that results from executing the code below.

```
def this(x):
    return 2*that(x)

def that(x):
    x = y + 1
    this = that
    return x

x, y = 1, 2
this(that(y))
```

Solution:



2.2 Lambdas

1. Fill in the blanks with one-line lambda expressions so that each call expression that follows returns 3.

```
>>> f1 = ____________________
>>> f1()
3
```

Solution:

```
lambda: 3
```

```
>>> f2 = ____________________
>>> f2()()
3
```

Solution:

```
lambda: lambda: 3
```

```
>>> f3 = ____________________
>>> f3()(3)
3
```

Solution:

```
lambda: lambda x: x
```

```
>>> f4 = ____________________
>>> f4()()(3)()
3
```

Solution:

```
lambda: lambda: lambda x: lambda: x
```

2.3 Lists and List Comprehension

1. Write a function that rotates the elements of a list to the right by k. Elements should not "fall off"; they should wrap around the beginning of the list. `rotate` should return a new list. To make a list of n 0's, you can do this: `[0] * n`

```
def rotate(lst, k):
    """ Return a new list, with the same elements
        of lst, rotated to the right k.
```

```
    >>> x = [1, 2, 3, 4, 5]
    >>> rotate(x, 3)
    [3, 4, 5, 1, 2]
    """
```

Solution:

```
    n = len(lst)
    ret = [0] * n
    for i in range(n):
        j = (i + k) % n
        ret[j] = lst[i]
    return ret
```

or

```
    return lst[-k:] + lst[:-k]
```

2. Define a function `foo` that takes in a list `lst` and returns a new list that keeps only the even-indexed elements of `lst` and multiplies each of those elements by the corresponding index.

```
def foo(lst):
    """
    >>> x = [1, 2, 3, 4, 5, 6]
    >>> foo(x)
    [0, 6, 20]
    """

    return [_____________________________________________________]
```

Solution:

```
    return [i * lst[i] for i in range(len(lst)) if i % 2 ==
        0]
```

3. Implement the functions `max_product`, which takes in a list and returns the maximum product that can be formed using nonconsecutive elements of the list. The input list will contain only numbers greater than or equal to 1.

```
def max_product(lst):
    """Return the maximum product that can be formed using lst
    without using any consecutive numbers
    >>> [10,3,1,9,2] # 10 * 9
    90
    """
```

Solution:

```
    if lst == []:
        return 1
    elif len(lst) == 1:
        return lst[0]
    else:
        return max(max_product(lst[1:]), lst[0]*max_product
            (lst[2:]))
```

2.4 Trees

1. An **expression tree** is a tree that contains a function for each non-leaf root, which can be either `'+'` or `'*'`. All leaves are numbers. Implement `eval_tree`, which evaluates an expression tree to its value. You may want to use the functions `sum` and `prod`, which take a list of numbers and compute the sum and product respectively.

```
def eval_tree(tree):
    """Evaluates an expression tree with functions as root
    >>> eval_tree(tree(1))
    1
    >>> expr = tree('*', [tree(2), tree(3)])
    >>> eval_tree(expr)
    6
    >>> eval_tree(tree('+', [expr, tree(4)]))
    10
    """
```

Solution:

```
    if is_leaf(tree):
        return entry(tree)
    args = [eval_tree(child) for child in children(tree)]
    if entry(tree) == '+':
        return sum(args)
    elif entry(tree) == '*':
        return prod(args)
```