DATA C88C Spring 2026

Higher-Order Functions, Environment Diagrams

Discussion 3: February 9, 2026

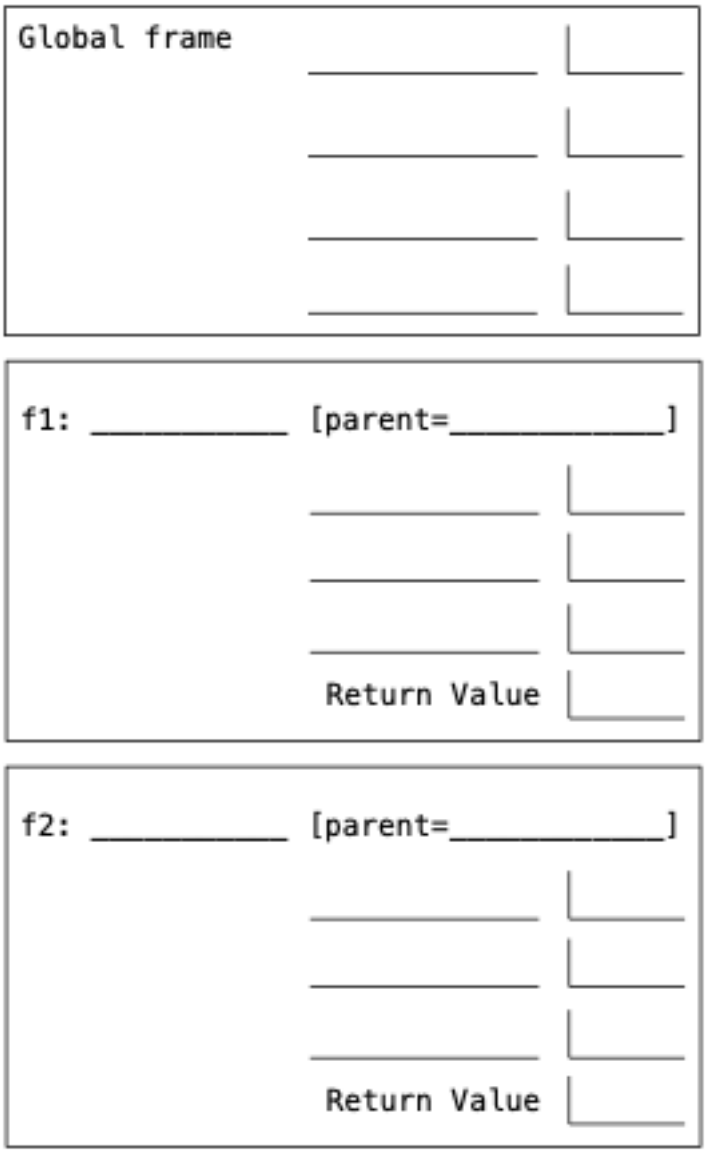
Environment Diagrams

Draw an environment diagram for the code below. You can use paper or a tablet or the whiteboard. Then, step through the diagram generated by Python Tutor to check your work.

```
def foo(x, y):
    foo = bar
    return foo(x, y)
def bar(z, x):
    return z + y
y = 5
foo(1, 2)
```

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

Here's a blank diagram in case you're using a tablet:



template

Note: This worksheet is a problem bank—most TAs will not cover all the problems in discussion section.

Higher-Order Functions

Remember the problem-solving approach from last discussion; it works just as well for implementing higher-order functions.

1. Pick an example input and corresponding output. *(This time it might be a function.)*
2. Describe a process (in English) that computes the output from the input using simple steps.
3. Figure out what additional names you'll need to carry out this process.
4. Implement the process in code using those additional names.
5. Determine whether the implementation really works on your original example.
6. Determine whether the implementation really works on other examples. (If not, you might need to revise step 2.)

Q1: Multi-Apply

Sometimes we want to apply a function more than once to a number. Implement multi-apply, which is a higher-order function takes in a function `f`. It returns a function of the form `g(x, y)`, which takes in two arguments. This new function *composes*, or applies, `f` to `x` `y` times; for example, for `y = 3`, it would evaluate `f(f(f(x)))`.

```
def multi_apply(f):
    """Returns a function g(x, y) that returns the result of applying f to x y times.

    >>> def adder(x):
    ...     return x + 1
    >>> multi_add = multi_apply(adder)
    >>> multi_add(3, 1)
    4
    >>> multi_add(4, 5)
    9
    >>> multi_add(5, 0)
    5
    """
    "*** YOUR CODE HERE ***"
```

Q2: Make Keeper

Implement `make_keeper`, which takes a positive integer `n` and returns a function `f` that takes as its argument another one-argument function `cond`. When `f` is called on `cond`, it prints out the integers from 1 to `n` (including `n`) for which `cond` returns a true value when called on each of those integers. Each integer is printed on a separate line.

```
def make_keeper(n):
    """Returns a function that takes one parameter cond and prints
    out all integers 1..i..n where calling cond(i) returns True.

    >>> def is_even(x): # Even numbers have remainder 0 when divided by 2.
    ...     return x % 2 == 0
    >>> make_keeper(5)(is_even)
    2
    4
    >>> make_keeper(5)(lambda x: True)
    1
    2
    3
    4
    5
    >>> make_keeper(5)(lambda x: False)  # Nothing is printed
    """
    "*** YOUR CODE HERE ***"
```