CS61A Midterm 1 Sheet Wang Jianzhi

Prep: bring ID, a pencil (for environment diagram), lots of paper, pen

You got this!

WWPD

`print(20, 20)` has a space between 20 20

`return None` does not show anything on IDE (!)

`return "hi"` gives `'hi'` (change to single quote)

`return 'hi'` gives `'hi'`

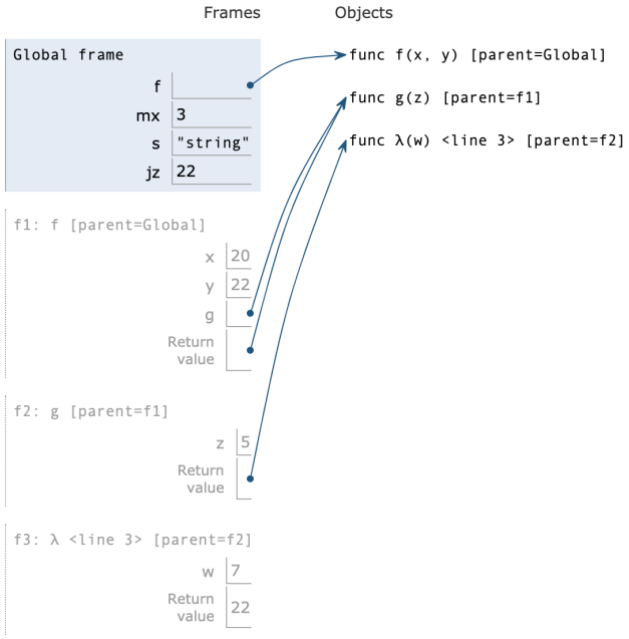
`print()` gives new line

Falsy values: `False, None, "", 0, 0.0, 0j, [], ()` …

All `Function` are Truthy.

```
>>> 1 == True
True
>>> 0 == False
True
```

Environmental Diagram Rules



All functions must be labelled with `[parent=…]` which is the frame they are defined at (**NOT** called).

Lambda functions are labelled with line number `<line …>`.

Function objects `funcs` are labelled with parameters `f(x, y)`.

`func max(…) [parent=Global]`

Evaluate every variable in the scope first, THEN open frame (!) Star the function, write out variables on a separate piece of paper before drawing a new frame. Don't get tricked by `z` in `f(g(x, y), z)`.

Check for any final assignments, particularly in the global frame.

Lambda Variants

- Currying: converting a function that takes multiple arguments into a single-argument HOF
- 2-recursive: outer function returns inner function which returns the outer function called on arguments. In the end, only the inner function is being repeatedly run. States can be memoized and chained in the child function

```
def detector(f):
      def g(i):
            # logic
            # calls parent function with complete arguments
            # i == x is memorized
            return detector(lambda x: f(x) or i == x)
      return g
```

- 3-recursive (typically consists of initiation function, then a 2-recursive)

```
def maxer(smoke):
      # initiation function
      def fire(y):
            # 2-recursive parent
            def haze(z):
                  # 2-recursize child
                  # logic
                  return fire(z);
            return haze
      return fire

def compose(n):
      # returns g(x) = f1(f2(…fn(x)…))
      if n == 1:
            return lambda f: f
      def call(f):
            def on(g):
                  return compose(n - 1)(lambda x: f(g(x)))
            return on
      return call
```

- Multiple lambdas

```
# Fall 2016
def multiadder(n):
      # >>> multiadder(3)(5)(6)(7)
      # 18
      if n == 1:
            return lambda x: x
      else:
            # essentially lambda, since one argument given
            return lambda x: lambda y: multiadder(n - 1)(x + y)

zipper = lambda x: x
helper = lambda f, g: lambda x: f(g(x))
zipper = helper(f1, zipper)
```

Final Checks

`n // 10 vs n / 10, float("-inf")`

`pow()` gives float when any argument is float, use `round()`

Thinking Process

- Straightforward solution (iteration)
- Get hints from previous functions in the same problem; Inspect arguments & return value
- Use lambda functions, double lambda; Remember that functions can take on values also
- Look at lambda variants above
- Recursion

- Whack, try test cases, whatever works
- Check and verify examples