Worksheet 2:

Higher Order Functions & Recursion

Higher Order Functions

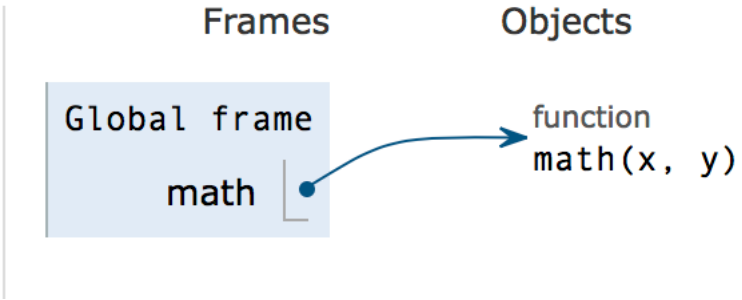
Data types may not just be integers or Booleans. They could be functions! We can use environment diagrams to model these higher-order functions.

Let's take an example:

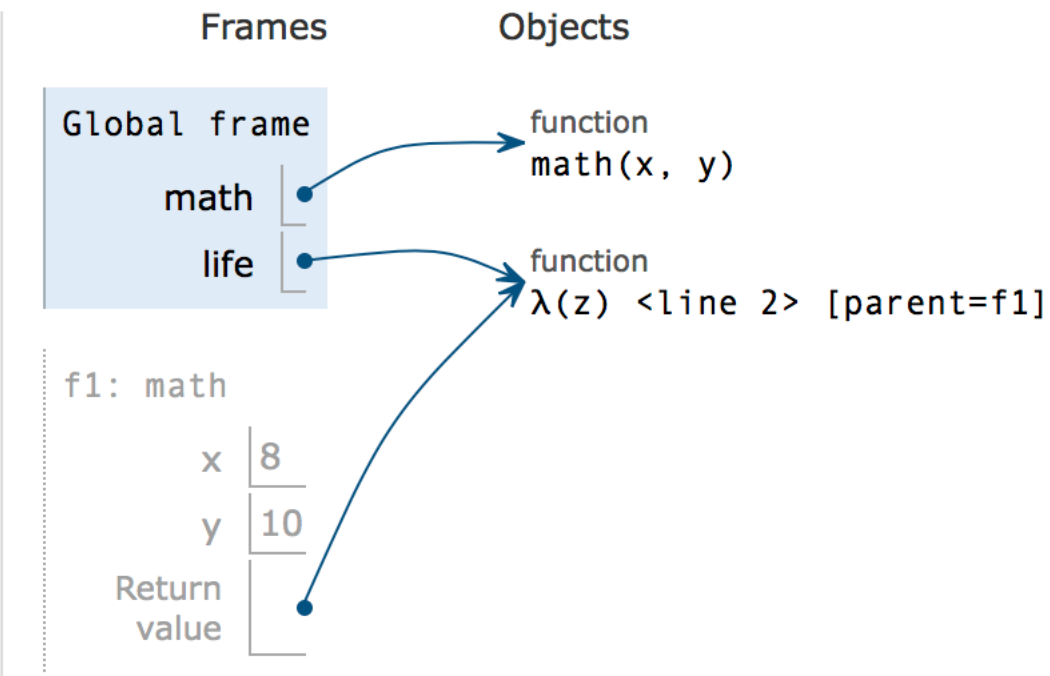
```
def math_is_life(x, y):
        return lambda z: 4 * y + x - z
life = math_is_life(8, 10)
life(6)
```

Here, we use a lambda statement as a function. Remember, a lambda function's parent is the frame in which it is **declared.**

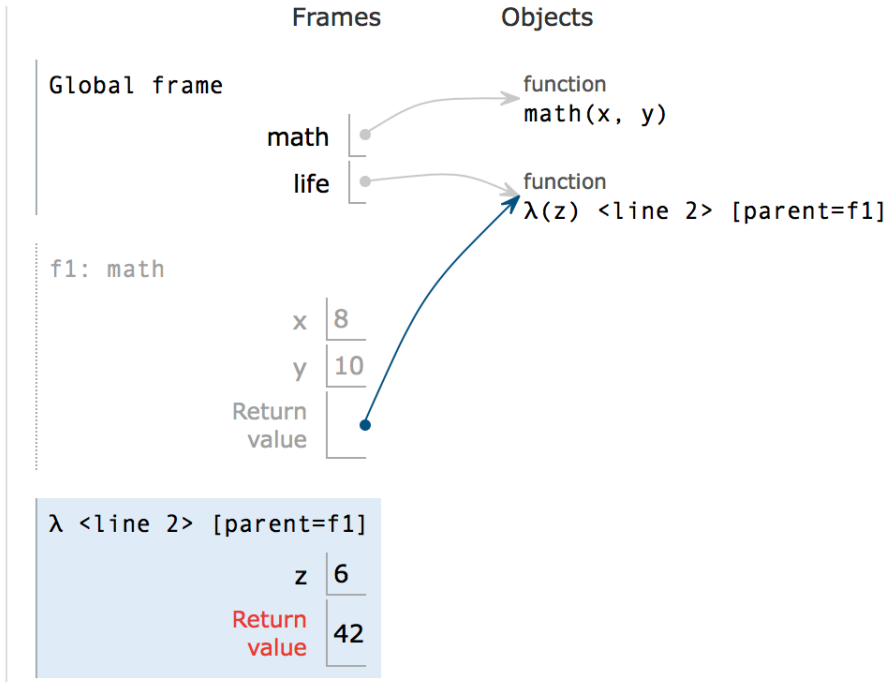
Let's dive into the environment diagram:



First, we declare the function.



Then, we open a new frame when we hit a function call. Here, we used the lambda symbol to represent a lambda function. Its parent is f1, since it was declared in f1. This function is then returned, and bound to the variable life. This is an example of a function being returned.



When we open our new frame for our lambda function, we assign the value 6 to z, but our return statement requires x and y! Since the lambda function doesn't have it, we might think this function Errors.

However, from the power of higher order functions, it actually returns 42.

If a frame uses a variable that's not declared in its frame, we look for that variable in the frame's parent.

Here, x and y are assigned in the lambda function's parent. So, we can use those values to get 42.

Let's do some practice problems:

Draw the environment diagram for the following problem:

Remember:

```
>>>'hello' + ' ' + 'human'
'hello human'
```

```
def water (x):
        adult = 'baby'
        y = lambda ice : x + x + adult
        def gas (z):
                z = z + z
                return y(z)
        return gas
gas = water('ice')
water = gas('gas')
```

Challenge:

```
def murph (murphy):
        murph = 'time'
        murphy = murph + ' ' + murphy
        def murph (murphy):
                murph = 10
                def murphy (murphy):
                        return murph + 20
                return murphy
        return murph('ys law')
murph = murph('love')
gravity = murph('eureka')
```

Code Writing:

Write a lambda function for the following function:

```
def make_add_mul(a):
        def make_add(b):
                def make_mul(c):
                        return a + b * c
                return make_mul
        return make_add
```

Complete the function:

```
def check(a):
        """Returns a function which takes one parameter b and returns 'yes' if
both a and b are True and 'no' otherwise
        >>> check(True)(True)
        'yes'
        >>> check(False)(True)
        'no'
        """
```

```
def loopy(n):
        """Returns a function which takes one parameter b and prints every
digit of b from right to left, for n digits. If # of digits in b < n, return
'done' after printing each digit. Otherwise, return b, missing the digits
printed.
        >>> four = loopy(4)
        >>> four(123456)
        6
        5
        4
        3
        12
        >>> four(12)
        2
        1
        'done'
        >>> zero = loopy(0)
        >>> zero(61)
        61
        """
        _____________________________________________

              _____________________________________________

              _____________________________________________

                    _____________________________________________

                    _____________________________________________

                    _____________________________________________

                          _____________________________________________

                    _____________________________________________

              _____________________________________________

        _____________________________________________
```

Recursion

This concept is one of the most difficult, but VITAL concept in this class. Recursion is when a function calls itself within its own body. When attempting recursive problems, we must keep a few things in mind.

1. **We need a BASE CASE**

A base case is usually the simplest input into the function. This will stop the function from going farther, allowing us to stop recursively calling it.

2. **Make the RECURSIVE call using a simpler input**

If we want to call our function, we need a different input, otherwise, we will continually and endlessly call the same thing repeatedly. We need to change our input, so that we eventually hit our base case. In addition, when we write our recursive call, we must assume that the recursive function works. This is called the "recursive leap of faith".

3. **Recursive Leap of FAITH**

When solving a recursive problem, we need to assume the recursive call works in order to solve the rest of the problem.

This might seem like a lot, so let's dive into a problem from lecture: factorial!

Review:

5! = 5*4*3*2*1

4! = 4*3*2*1

0! = 1! = 1

Now, let's implement this recursively,

```
def factorial(n):
        if ____________________________:
                return ____________________________
        else:
                return ____________________________
```

First, we need a base case.

We know the simplest possible input is either 1 or 0, as both of those return 1. Therefore, it might be a good idea to use n==0 or n == 1 as our base case. This way, we know our function will stop here.

```
def factorial(n):
        if (n == 0 or n == 1):
                return 1
        else:
                return ____________________
```

Now, we need our recursive case.

Let's try to make our problem *simpler*. For example, we can generalize 5! to 5*4!

If we work under the assumption our factorial function works, we can write our recursive case!

```
def factorial(n):
        if (n == 0 or n == 1):
                return 1
        else:
                return n*factorial(n-1)
```

And we are finished!

Your turn!

Code Writing Questions

```
def count_digits(n):
        """Returns the number of digits in the inputted number. Implemented
recursively!
                >>> count_digits(43345)
                5
                >>> count_digits(123456)
                6
                >>> count_digits(0)
                1
                """
        if ____________________:

                return ________________________
        else:
                return ________________________
```

Keep indentations!

```
def print_every_other(n):
        """Print every other number of digits (backwards)in the inputted
number. Implemented recursively!
                >>> print_every_other(43345)
                5
                3
                4
                >>> print_every_other(123456)
                6
                4
                2
                >>> print_every_other(0)
                0
                """
        def helper(_________________, _________________):

                ____________________________

                        ____________________________

                        ____________________________

                ____________________________

                        ____________________________

                ____________________________

                        ____________________________

                ____________________________

        ___________________________________
```